Le deuxième chapitre présente tous les concepts de base du langage, et vous permettra de vous familiariser avec le dictionnaire, les vocabulaires, la notation polonaise et l'utilisation des piles.

Dans le troisième chapitre, nous entrons dans la partie technique de programmation où vous apprendrez à définir des mots et à les exécuter, en utilisant toutes les structures de contrôle qui vous sont fournies dans le FORTH de base.

L'objet du quatrième chapitre est de vous présenter deux vocabulaires classiques: l'EDITEUR et l'ASSEMBLEUR.

Le chapitre cinq, quant à lui, va vous initier à la mécanique de fonctionnement des langages tissés. La lecture de ce chapitre s'adresse à des lecteurs non débutants en FORTH, car il est souvent fait appel à des références au langage lui-même. D'une bonne connaissance de ce fonctionnement on pourra tirer une meilleure qualité des programmes, et une utilisation optimisée des outils.

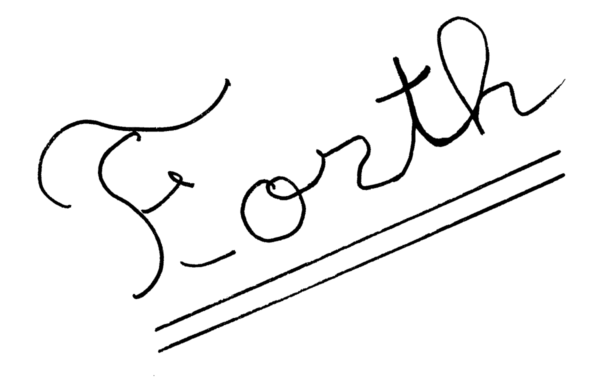
Le chapitre six présente tous les outils de haut niveau dont FORTH est muni. Définir de nouvelles structures de données adaptées aux problèmes à résoudre, et de nouvelles structures de contrôle, voici quelques exemples d'application de ces mots dits de haut niveau.

Au cours du chapitre sept, nous nous intéresseront de près à certains aspects de FORTH: Vocabulaires, Segmentation, Recursivité, Multitasking.

Un recueil de problèmes avec leur solution commentée se trouve dans le huitième et dernier chapitre.

A la fin de chaque chapitre, le lecteur trouvera un tableau récapitulatif de tous les mots nouvellement rencontrés, ainsi qu'une sélection d'exercices simples, destinés à mettre immédiatement en pratique les nouvelles notions acquises.

TABLE DES MATIERES



REMERCIEMENTS

Nous tenons à remercier plus particulièrement M.S.SOLOTAREFF pour son apport documentaire, ROBOTIQUE S.A. pour sa participation constructive et les moyens techniques mis à notre disposition, ainsi que M. TCHOUMAKOFF pour ses recherches assidues de nouvelles publications.

Nous n'oublierons pas Jean-Michel BARNAY pour son dénigrement systématique, ainsi qu'EDITH pour son Chili con carne et NICOLE pour son soutien dactylographique.

Les Auteurs

I

INTRODUCTION ET GÉNÉRALITÉS

I.1. SITUATION HISTORIQUE

FORTH est l'œuvre d'un seul homme : Charles H. MOORE. Sa première implantation complète a été effectuée sur un IBM 1130 à l'époque des ordinateurs de troisième génération (fin des années 60).

Le résultat de son travail fut suffisamment satisfaisant pour que MOORE n'hésite pas à l'appeler langage de quatrième génération : FOURTH. Malheureusement, l'IBM 1130 ne tolérait que des noms de programme de cinq lettres, ce qui a décidé du nom définitif du langage : FORTH.

Après avoir testé plusieurs parties de FORTH sur différentes machines (Burroughs 5500, IBM 360) et en différents langages (ALGOL, COBOL), FORTH a été codé en FORTRAN avant de l'être en ASSEMBLEUR, puis en FORTH.

La première application de FORTH a été la programmation d'un terminal graphique IBM 2250 connecté à un IBM 1130.
Un programme FORTRAN de 32 Koctets permettait de tracer de modestes graphiques plans : une application FORTH de 8 Koctets fournissait des dessins tridimensionnels en mouvement !

La première application professionnelle (1971) développée en FORTH par Charles MOORE a été un problème d'acquisition de données de radio-téléscope au National Radio Astronomy Observatory sur un Honeywell 316. Cette application a ensuite été reprise au NRAO de Kitt Peak (Tucson) sur un PDP 11. Ce radio-téléscope à ondes millimétriques a

permis au cours des dix dernières années de découvrir la moitié des molécules interstellaires connues.

En 1973, Charles MOORE crée sa société de développement de logiciels FORTH : FORTH Inc.

Le marché de l'astronomie étant trop limité, d'autres applications ont été vite mises au point :

— MiniFORTH : FORTH pour mini-ordinateur.
— Multiprogrammation en FORTH.
— Système de gestion de bases de données en FORTH.
— PolyFORTH : un "super"FORTH.
— MicroFORTH : FORTH pour micro-ordinateur.

En 1976, un comité de l'International Astronomical Union adopte FORTH comme standard de langage de programmation pour l'astronomie.

Le nombre des adeptes de FORTH dans la communauté informatique mondiale va croissant de jour en jour. Ce phénomène a suscité la création d'un FORTH Interest Group aux États-Unis dont la vocation est de faire connaître ce langage. C'est par exemple lui qui distribue la norme choisie par l'International FORTH Standard Team (La dernière norme date de 1979).

C'est cette norme qui constitue le noyau de la plupart des FORTH commercialisés et qui permet la transportabilité des logiciels.

Dans l'industrie de la programmation, un nouveau langage n'aura une chance d'être adopté que s'il apporte des solutions aux problèmes de la réduction des coûts et de la fiabilité (au sens large) des logiciels. L'expérience a montré que l'esthétique ne peut être retenue comme critère de choix.

Le développement frénétique de la micro-informatique et la chute continue des coûts de production des matériels, font que les constructeurs cherchent à implanter des logiciels de base modernes, performants et faciles à utiliser pour un coût aussi faible que possible.

Actuellement, PASCAL répond à ces exigences mais nécessite une configuration assez lourde (48 Koctets + mémoire de masse) pour un budget grand public.

Dans le domaine professionnel, on attend de l'informatique la résolution de problèmes souvent très complexes. Les langages de programmation utilisés jusqu'à maintenant imposent des coûts de développement et de tests tels que l'investissement et le plan informatique peuvent être remis en cause.

Pour juger de l'avenir d'un nouveau langage de programmation, il ne faut pas juger simplement sur les aspects techniques, mais en termes de mar-

[illegible]é. D[illegible] l'évo[illegible]n de [illegible]RTH [illegible] cou[illegible] ces [illegible]nière[illegible] [illegible]nées [illegible] t porte à croire qu'il résout efficacement les problèmes de tout ses utilisateurs actuels dont la population ne cesse de croître.

Les constructeurs quant à eux adoptent de façon de plus en plus systématique FORTH comme langage de base sur leur matériel. Quelques machines ont même été conçues pour FORTH. Dans certains secteurs d'activité (ludotique, astronomie, etc...) FORTH est devenu l'outil privilégié de développement.

FORTH ne s'inscrit pas dans la philosophie des langages traditionnels, il est donc intéressant de situer ses caractéristiques par rapport à ces derniers.

I.2. FORTH ET LES LANGAGES INFORMATIQUES TRADITIONNELS

Un langage informatique n'est qu'un outil comme un autre, et doit donc être fonctionnel.

Un bon outil doit être facile à utiliser. Par exemple le grand succès de BASIC tient à des raisons pédagogiques : aucune connaissance théorique n'est indispensable.

Notre objectif dans cet ouvrage est de vous montrer que l'apprentissage de FORTH est à la portée de tous :

— la syntaxe est simple et non contraignante,

— l'utilisateur définit ses propres mnémoniques et choisit donc son vocabulaire de dialogue avec la machine.

Pendant le développement d'une application, il est agréable d'avoir un contrôle permanent de la syntaxe, c'est la raison d'être des interpréteurs. Cet agrément se paie au niveau du temps d'exécution. L'idéal serait d'avoir la vitesse d'exécution d'un langage compilé et l'aide au développement d'un interpréteur.

FORTH réunit ces deux qualités en étant un langage semi-compilé.

Les langages évolués n'offrent jamais d'outil de manipulation de données au niveau du bit en mémoire. C'est une des raisons qui oblige à écrire des modules externes en assembleur. Bien qu'étant un langage de haut niveau le FORTH standard permet ces manipulations de données. Il peut inclure un vocabulaire spécialisé ASSEMBLEUR qui permet d'utiliser des mnémoniques assembleurs tout en restant en FORTH. Certaines machines

permettent de faire appel à des routines assembleur dans un programme écrit dans un langage évolué, mais ces routines doivent être développées séparément et doivent être résidentes en mémoire au moment de l'appel.

Ce qui caractérise la génération d'un langage informatique de haut niveau est la présence ou l'absence de structuration. Les structures de contrôle sont indispensables à une programmation éclairée. C'est paradoxalement la grande lacune des langages les plus répandus comme COBOL, FORTRAN, BASIC.

Comme avec PASCAL, FORTH offre les structures de contrôle classiques du type BEGIN ––– WHILE ––– REPEAT. Il va même beaucoup plus loin puisqu'il permet à l'utilisateur de créer ses propres structures de contrôle et de données.

On n'insistera jamais assez sur l'importance des structures de données dans la programmation. Une bonne analyse de problème ne fait que dégager parmi un ensemble de données en vrac, des caractéristiques de structures et de gestion de ces structures.

Les langages traditionnels n'offrent au programmeur qu'une liste figée de structures de données possibles et son problème sera donc de faire le choix le moins mauvais pour se rapprocher des structures naturelles issues de son analyse.

Le défaut classique des analystes est de raisonner par rapport à leur langage de travail et de ne plus savoir extraire les structures naturelles.

Le conflit structures de données / langage a été apaisé provisoirement avec l'apparition des systèmes de gestion de base de données. Ils permettent d'entretenir des structures complexes de données mais sont généralement autonomes. Les interfaces avec les langages, lorsqu'elles existent, sont très lourdes à manipuler et le programmeur est contraint à passer plus de temps à préparer et à contrôler les accès qu'à traiter effectivement les données.

En FORTH, les structures de données naturelles extraites de l'analyse, quelqu'elles soient, pourront être implantées directement et le programmeur se construira tous les modes d'accès et de contrôle de cohérence qu'il juge nécessaire.

FORTH est un langage adulte, ce n'est pas lui qui vous impose une liste de contrôles, dont l'expérience a montré qu'elle était toujours insuffisante. Il laisse le programmeur seul juge et responsable du niveau de sécurité à apporter au logiciel.

Il est enfin possible de créer des applications à sécurité totale.

... FORTH ET LES MACHINES

FORTH, aussi bizarre que cela puisse paraître pour les non initiés, peut être écrit en FORTH pour sa quasi-totalité. La faible taille du noyau actif qui doit être écrit dans un langage déjà implanté, (moins de cent lignes d'assembleur par exemple), permet à toute personne bien imprégnée de la philosophie FORTH de développer seule un FORTH standard.

Sa méthode de travail et les problèmes qu'elle sera amenée à résoudre seront identiques à ceux de toutes les applications qu'elle souhaitera développer ultérieurement en FORTH : il n'y a pas de discontinuité entre la création et l'utilisation de FORTH. Nous espérons que vous-même aurez pris suffisamment goût à cette nouvelle philosophie conviviale pour implanter FORTH sur votre machine préférée.

Nous allons maintenant vous fournir une liste des implantations de FORTH. Celle-ci n'a pas la prétention d'être exhaustive. Nous venons en effet de voir que l'implantation du langage était un problème simple à résoudre, et tout à fait à la portée d'un seul programmeur. Ceci rend impossible le suivi des différentes implantations réalisées.

Nous désirons seulement vous faire percevoir l'éventail des matériels actuellement atteints par cette nouvelle philosophie.

— *Les micro processeurs 8 bits*

- Intel : 8080, 8085, 8088
- Zilog : Z80
- Motorola : 6800, 6809
- RCA : 1802
- Rockwell : 6502

— *Les micro processeurs 16 bits*

- Intel : 8086, IAPX 186, IAPX 286
- Zilog : Z 8000
- Motorola : 68000
- Digital Equipement : LSI 11
- Texas Instruments : 9900

— *Les micro-ordinateurs*

- APPLE II
- HP 85, HP 9826, HP 9836
- TRS 80

- ZX
- Atari 800
- Commodore VIC 20
- Panasonic HHC
- PET
- IBM Personal Computer
- Heathkit 89 / Zenith 89
- Northstar
- Cromenco
- Jupiter Ace
- SMS 300
- Goupil

— *Autres ordinateurs*

- PDP 11, PDP 10, PDP 8
- NOVA
- HP 2100
- Burroughs 5500
- IBM 360, IBM 1130, Serie 1
- Univac 1108
- Mod-Comp II
- CDC 6400
- Interdata
- Illiac
- Honeywel 316, MINI 6
- Varian 620
- GA SPC-16
- Phase IV
- Computer Automation LSI 4

I.4. QUELQUES APPLICATIONS

FORTH est très utilisé dans les effets spéciaux cinématographiques. A ce titre, la société Elican Inc (Brea - CA) emploie maintenant FORTH pour commander les systèmes de caméra du type de celles utilisées pour le film Star Wars. Ce système a de plus été utilisé par New World Productions (Venice - CA) pour le tournage du film Battle Beyond The Stars. Citons aussi la société Magican Inc, qui a réalisé les effets spéciaux de Star Treck, et qui est maintenant équipée de FORTH.

Pour rester dans le domaine des applications "divertissantes", citons l'expérience d'Atari, qui a développé un système de tests pour les jeux électroniques de café utilisant FORTH sur un microprocesseur 6502. Les nouveaux jeux de ce type apparaissant sur le marché fonctionneront sans doute très prochainement en FORTH. De plus, Atari développe ses jeux personnels en FORTH, et offre d'ailleurs sur son micro-ordinateur personnel Atari 800 une version de FORTH appelée Game FORTH.

Le fameux hôpital Cedar Sinai Medical Center de Los Angeles utilise FORTH sur un PDP 11/60 pour accomplir simultanément les tâches suivantes :

— gestion de 32 terminaux,

— acquisition de données sur les patients à l'aide d'un lecteur optique, et classement des informations dans une importante banque de données,

— traitements statistiques sur cette banque de données,

— analyse en temps réel des échantillons sanguins et des informations cardiaques d'un patient.

De plus, cet hôpital développe actuellement un système portable de surveillance des malades, à l'aide d'un micro-processeur 6800, toujours en FORTH.

Le traducteur de poche CRAIG M100, qui offre la possibilité de traduire entre elles une vingtaine de langues, suivant le module choisi, a été développé en FORTH.

Dans le domaine des applications spatiales, nous en citerons deux, en plus des radio-télescopes dont nous avons déjà parlé.

Une version spéciale de FORTH, appelée IPS, est utilisée dans un satellite de radio amateurs : OSCAR Phase III.

La société Avco Inc. utilise FORTH sur un 1802 pour la surveillance des températures sur un satellite militaire ainsi que pour les échanges de données télémétriques.

Encore une fois, cette liste n'est en aucun cas exhaustive. Elle présente seulement l'intérêt de faire percevoir la diversité des applications de FORTH (Gestion, contrôle de processus, calcul scientifique, etc...).

I

PRÉSENTATION DU LANGAGE

Considérons le cas de Monsieur J-M B intéressé par la micro informatique, qui vient d'acquérir un micro-ordinateur personnel, et qui découvre pour la première fois un langage de programmation évolué du type BASIC.

Il est fort probable que les premiers pas dans le logiciel de Monsieur J-M B se traduiront par des programmes labyrinthes, pavés compacts d'instructions enchevêtrées. Pour la mise au point, il aura selon toute vraisemblance fait appel à des outils type TRACE, qui seuls peuvent lui donner une chance de suivre le déroulement de ses exécutions.

Monsieur J-M B comprendra très vite que cette méthode instinctive de programmation est inadaptable aux problèmes complexes, et ne peut fournir que des produits à courte durée de vie, toute retouche ultérieure se révélant plus longue que la réécriture complète du logiciel lui-même.

Les torts de Monsieur J-M B sont manifestement de ne pas avoir entrepris une analyse descendante poussée, et d'avoir improvisé face aux cas non prévus.

Il ne faudrait pas croire que Monsieur J-M B est le seul responsable de ses errements. Les langages classiques encouragent une programmation au coup par coup, prônant les branchements comme issue universelle aux mauvaises surprises inévitables après une analyse insuffisante.

Monsieur B L a une expérience plus longue de la programmation pour des raisons professionnelles. Il a déjà surmonté les problèmes de Monsieur J-M B, et a su dégager une méthodologie.

Il analyse les problèmes en profondeur, sait dégager les structures de données, et découpe sa programmation en blocs indépendants. Il relie enfin ces blocs entre eux grâce aux structures de contrôle dont il dispose.

Il rencontre moins de problèmes de mise au point et documente ses travaux car il sait prévoir des modifications ultérieures.

Ses compétences sont reconnues, mais tout problème nouveau l'oblige à repartir d'une mémoire vierge. Il sait dégager des blocs, mais ne sait pas faire le rapprochement entre blocs de même nature, pour construire les modules à usage général.

Les temps de développement restent longs, car il ne sait pas avoir une approche globale des problèmes et se doter d'outils réutilisables.

Monsieur G M, quant à lui, a oublié son passage par les étapes J-M B et B L. Il se refuse désormais par paresse à entreprendre un programmation fastidieuse. Sa force et son efficacité résident dans le fait qu'il sait généraliser ses procédures pour pouvoir les réemployer aisément et à bon escient lors d'analyses nouvelles.

Il souffre toutefois de ne pouvoir les intégrer dans la bibliothèque du langage, afin de rendre leur emploi encore plus immédiat.

Son rêve serait d'avoir la possibilité d'étendre son langage de travail en fonction de l'application.

II.1. PREMIER CONTACT

II.1.1. Concept de MOT.

Jusqu'alors, le programmeur devait se soumettre aux contraintes du langage avec lequel il communiquait en lui proposant des instructions.

Avec FORTH, il peut enseigner son langage à la machine : le dialogue s'instaure enfin.

Un programme traditionnel consiste en un enchaînement d'instructions cataloguées.

En FORTH, le dialogue avec la machine se fait en lui communiquant des suites de mots. L'utilisateur peut décider de donner un nom à une suite quelconque de mots que la machine reconnaîtra alors comme nouveau mot du langage.

Il va de soi que FORTH reconnaît déjà une liste importante de mots effectuant aussi bien des actions élémentaires que des tâches élaborées.

La notion de programme, et celle d'instruction, n'existe plus. Le travail

de programmation consiste à construire ou étendre un vocabulaire en définissant de nouveaux mots à partir des mots connus de la machine.

II.1.2. Concept de DICTIONNAIRE.

Comme l'être humain, la machine dispose d'un dictionnaire qu'elle enrichit par l'apprentissage de nouveaux mots.

Comme dans toutes les langues, un dictionnaire est un objet amorphe et volumineux. Il est naturel de l'organiser en vocabulaires de mots ayant des traits communs, libres et subjectifs. Pour vous en convaincre, essayez d'extraire rapidement d'une encyclopédie en vingt volumes tous les mots ayant trait à l'Informatique...

FORTH offre cette possibilité de raisonner en termes de vocabulaires.

Lorsque la machine se voit proposer un mot, elle va chercher à savoir si elle le connait en explorant le vocabulaire que vous avez choisi.

Tout apprentissage se fait bien sûr à partir des connaissances acquises, ce qui signifie que pour apprendre un mot nouveau à votre machine, vous ne pourrez utiliser que des mots qu'elle connait déjà dans le vocabulaire de travail.

L'apprentissage terminé, la machine possède alors une définition du nouveau mot à laquelle elle se réfèrera.

Par abus de langage, nous appellerons également définition cette phase d'apprentissage.

Bien entendu, il peut vous arriver de changer d'avis et de redéfinir un mot existant. Lorsque FORTH a besoin de reconnaître un mot, il se réfèrera toujours à la définition la plus récente qu'il en connait.

Toute nouvelle définition d'un mot ne sera pas prise en compte par les mots qui faisaient appel à l'ancienne définition de ce mot.

Exemple : Le mot POLITESSE est défini à l'aide du mot BONJOUR.
Le mot SALUTATION est défini à l'aide du mot POLITESSE.

SALUTATION utilise donc indirectement le mot BONJOUR.

Si le mot POLITESSE est redéfini avec le mot AU-REVOIR, le mot SALUTATION continuera tout de même à utiliser l'ancienne version de POLITESSE, et donc le mot BONJOUR.

Nous avons tenu à signaler tout de suite ce qui peut apparaître comme un premier obstacle au débutant en FORTH, du fait de sa culture informatique classique. Mais nous verrons que non seulement ce problème est très facilement soluble, mais que de plus la philosophie du FORTH peut en faire un avantage.

Baptême de l'air :

Parmi les mots de base, FORTH offre la possibilité d'effacer les derniers mots définis : c'est le mot FORGET.

Exemple : Supposons que les mots HEURE, MINUTE et SECONDE soient dans l'ordre les derniers définis.

La syntaxe : FORGET HEURE [Return] OK

effacera le mot HEURE du dictionnaire, ainsi que tous les mots définis postérieurement à HEURE, c'est-à-dire les mots MINUTE et SECONDE.

Exemple : C'est toujours la dernière définition du mot qui suit FORGET qui sera prise en compte.

Ainsi si l'on définit successivement HEURE, MINUTE, SECONDE, puis à nouveau HEURE,

FORGET HEURE [Return] OK

ne supprime que la dernière définition de HEURE.

II.1.3. Définition d'un MOT.

En FORTH, tous les mots doivent être séparés par un espace blanc : c'est l'unique séparateur.

Toute définition simple de mot commencera par le mot : (c'est le signal de début d'apprentissage), suivi du nom du mot que l'on définit, et se terminera par le mot ; (c'est le signal de fin d'apprentissage).

Exemple : Reprenons l'exemple précédent. Il s'écrit :

```
: POLITESSE BONJOUR ; [Return] OK
: SALUTATION POLITESSE ; [Return] OK
```

Le mot FORTH (indique que le texte qui suit est un commentaire, qui se terminera avec le caractère). Les commentaires peuvent apparaître à tout moment dans le dialogue : en cours de définition ou non.

oter t de que ise e ge d te es e du de la nature du séparateur.

Exemple : Les deux définitions suivantes sont équivalentes :

```
: TOTO TITI TATA ; [Return] OK

: TOTO      ( Définition du mot TOTO)  [Return]
  TITI      ( Action ... )  [Return]
  TATA      ( Action ... )  [Return]
; [Return] OK
```

II.1.4. Les objets du FORTH.

En FORTH, les données manipulées peuvent être des variables, des constantes, ou des constantes immédiates. Précisons tout de suite que ces objets sont de nature différente à l'intérieur même de la machine. Ainsi la valeur d'une constante n'est pas modifiable de façon immédiate.

Pour déclarer et initialiser une variable, la syntaxe est la suivante :

Valeur VARIABLE nom de la variable

Pour les constantes, la syntaxe est similaire :

Valeur CONSTANT nom de la constante

Les constantes immédiates sont des valeurs numériques volatiles.

Nous verrons plus loin la manipulation de ces objets. Notons cependant tout de suite que le FORTH n'est absolument pas figé sur les structures de données. Nous expliquerons à quel point la définition de structures telles que les chaînes de caractères, les matrices, ainsi que de toutes les structures personnalisées est aisée et rapidement mise en œuvre.

II.1.5. La structure de pile LIFO.

FORTH est construit autour de deux piles : La pile de données (DATA STACK) et la pile de retour (RETURN STACK).

Cette notion de pile est fréquemment rencontrée dans la vie courante. Ainsi une pile d'assiettes qui n'est alimentée que par le haut présente une structure telle que la première assiette posée sera la dernière retirée, et qu'inversement la dernière assiette posée sera la première retirée.

C'est précisément cette dernière caractéristique, à savoir que le dernier 13

...ré est le premier sorti, (Last In First Out), qui définit la structure de donnée dite de pile LIFO.

Bien que transparente pour l'utilisateur, cette structure est utilisée dans la résolution de nombreuses fonctions de base des ordinateurs : calculs d'expressions arithmétiques, appels de sous-programmes, analyse syntaxique, etc...

L'implantation d'une structure de pile dans une machine se fait de la manière suivante :

Les informations sont enregistrées séquentiellement à partir d'une adresse donnée qui est conservée dans un registre particulier. Les ajouts et suppressions se font par rapport à une autre adresse spécifiée dans un deuxième registre particulier qui fournit en permanence la première cellule disponible : C'est le pointeur de pile.

Ces deux registres sont initialisés à une même valeur. La condition d'égalité correspondra toujours à une pile vide. Il est facile d'en contrôler l'expansion en fixant sa capacité dans un troisième registre particulier. La condition d'égalité entre le pointeur de pile et ce troisième registre correspondra à une pile pleine.

L'utilisateur peut très facilement empiler puis dépiler des données manuellement dans la pile de données. Par exemple, le mot . extrait le sommet de cette pile et l'affiche à l'écran.

Ainsi, vous pouvez exécuter :

12 [Return] OK

24 [Return] OK

33 [Return] OK

Puis :

. [Return] 33 OK

. [Return] 24 OK

. [Return] 12 OK

Les données ont bien été reproduites dans l'ordre inverse de leur introduction.

L'exécution d'un point supplémentaire provoquera l'apparition d'un message d'erreur signalant que la pile de donnée est vide.

Tout est donc fort logique.

Comment pourtant expliquer que l'interpréteur ait accepté les mots 12, 24 et 33 qui n'ont jamais été définis ?

Revenons pour un instant sur la mécanique de l'interpréteur.

Il entre en activité dès que vous avez pressé la touche Return, et analyse le contenu de votre phrase. Il isole facilement les mots grâce à la présence du séparateur blanc. Chaque mot donne lieu d'abord à une recherche dans le dictionnaire. S'il est trouvé, la machine effectue le traitement nécessaire.

Ici, par contre, 12 ne figure pas dans le dictionnaire. L'interpréteur va donc maintenant tenter de l'identifier à une constante immédiate, c'est-à-dire un nombre exprimé dans la base courante. Notons en effet que FORTH est capable de travailler dans une base quelconque, comme nous le verrons plus loin. Si le nombre est valide, il sera placé au sommet de la pile, sinon un message d'erreur apparaîtra.

II.1.6. La notation polonaise inversée

Présentation

Lorsque vous utilisez votre calculette, les opérations sont directement exécutées dans l'ordre et au fur et à mesure de l'introduction des données.

Ainsi : 1 + 2 * 3 vaudra 9 .

Si vous voulez lui faire effectuer 1+(2*3), vous devrez nécessairement modifier l'ordre d'introduction et taper :

2 * 3 + 1 , ce qui donnera bien cette fois 7 .

C'est donc l'ordre d'introduction qui conditionne le résultat du calcul, les opérateurs étant tous banalisés.

Sur un ordinateur, la méthodologie est différente. Il n'interprète vos commandes qu'à la réception d'une marque de fin, par exemple le fait de presser la touche Return. Il analyse alors l'expression, et peut l'exécuter dans un ordre non séquentiel.

Si on tape A+B*C, la plupart des langages effectueront d'abord B*C, puis ajouteront A. On leur a en effet enseigné une hiérarchie des opérateurs. L'ordre classique des opérateurs courants est le suivant :

. ^ (Puissance), – (Négation), * et /, + et –, ...

C'est l'ordre séquentiel de gauche à droite dans l'expression qui regit les opérateurs de même niveau.

Cette hiérarchie pose un problème si l'on veut faire exécuter d'abord un opérateur de priorité inférieure. Par exemple, si dans A+B*C on souhaite additionner A et B d'abord, puis multiplier le résultat par C. C'est la fonction des délimiteurs (et) .

Il faudra donc écrire (A+B)*C.

L'automate de reconnaissance est alors plus élaboré, et les manipulations deviennent plus complexes.

Notation polonaise

Imaginons une notation qui ne modifie pas l'ordre des opérandes, et dont l'exécution est toujours déroulée de gauche à droite. Ce sens gauche → droite a été choisi arbitrairement.

Un tel type de notation permettrait:

— d'éliminer les hiérarchies, donc les parenthèses,

— de réduire l'automate à la simple reconnaissance des opérandes et des opérateurs.

Le principe de l'exécution serait alors le suivant:

— tous les opérandes sont mémorisés au fur et à mesure de leur rencontre,

— les opérateurs sont exécutés immédiatement. Ils opèrent sur les deux derniers opérandes mémorisés, qui sont alors détruits, et le résultat est rangé en mémoire comme un nouvel opérande.

Cette gestion des opérandes s'inscrit parfaitement dans la structure de pile LIFO décrite précédemment.

Notons que tous les interpréteurs arithmétiques classiques mettent toujours les expressions sous cette forme avant de les exécuter.

On convient de noter ↑ l'opération d'empilage et ↓ l'opération de dépilage.

Exemple:

A+B, noté A B + s'exécutera comme suit:

A: ↑
B: ↑
+: ↓ ↓ (A+B) ↑

16

(A+B)*C, noté A B + C * , s'exécutera:

A: ↑
B: ↑
+: ↓ ↓ (A+B) ↑
C: ↑
*: ↓ ↓(A+B)*C ↑

A+B*C, noté A B C * + , s'exécutera:

A: ↑
B: ↑
C: ↑
*: ↓ ↓(B*C) ↑
+: ↓ ↓ A+(B*C) ↑

On remarque qu'il y a toujours un empilage de plus que de dépilages, puisque le résultat final restera sur la pile.

Opérations arithmétiques

Sur votre terminal FORTH, vous pouvez appliquer tout de suite les notions qui viennent d'être présentées.

Tapez:

1 2 + [Return] <u>OK</u>

Puis, pour afficher le résultat:

. [Return] <u>3 OK</u>

N'oublions pas que tous les opérateurs arithmétiques sont des mots FORTH au même titre que les autres. A partir de maintenant, nous ne parlerons donc plus que de mots arithmétiques. Rappelons que ceux-ci vont toujours chercher leur opérandes sur la pile, et y rangent leurs résultats.

II.1.7. Les conventions de notation

Dialogue avec la machine

Toutes les touches de contrôle sont représentées dans un encadrement, par exemple:

[Return] , [Ctl] , [Esc] , [Break] , [Shift]

Toutes les réponses de la machine seront soulignées.

D , s, F 'H r la m... à l'u... teur ... affic... OK.

Représentation de la pile

On représente la pile entre parenthèses, en figurant de gauche à droite les données dans leur ordre d'entrée, suivies de ———.

Par exemple, après avoir effectué :

1 [Return] OK

2 [Return] OK

On représentera la pile comme suit:

(1 2 ———)

Représentation de l'action des mots FORTH sur la pile

On figure, entre parenthèses et séparés par ———, les paramètres sur la pile utilisée par le mot, et ceux éventuellement retournés après exécution.

Exemple : + (n1 n2 ——— n1+n2)

. (n ———)

II.1.8. La semi compilation

Approche interprétée

En mode interprété, les lignes de programme sont immédiatement analysées. Les mots clés, une fois reconnus sont remplacés par leur codes (Tokens).

A l'exécution, lorsqu'un de ces tokens est rencontré, le programme le recherche dans une table qui lui fournira l'adresse du code exécutable correspondant.

Approche compilée

L'utilisateur rédige d'abord le texte de son programme sous un éditeur.

Il appelle ensuite le compilateur qui fait une analyse globale du texte, et remplace directement les mots clés par du langage machine directement exécutable.

Approche semi compilée

Le programmeur dialogue ligne par ligne avec un interpréteur.

Ce dernier remplace les mots-clés, non pas par des tokens, mais directement par l'adresse de l'exécution du mot clé en question.

Cette dernière solution concilie les deux aspects précédents, souplesse de programmation, et rapidité d'exécution.

La semi-compilation existe dans la plupart des langages. Cependant ceux-ci ayant une bibliothèque de mots clés figée, les programmes sont nécessairement verbeux, l'utilisateur n'ayant qu'une gamme finie d'outils à sa disposition.

En FORTH, l'extensibilité du langage permet de définir des mots compacts et optimisés, et la semi-compilation trouve alors toute sa puissance.

II.2. LE VOCABULAIRE FORTH

Comme tout vocabulaire, le vocabulaire FORTH contient la liste des mots FORTH. Il n'y a pas de classement selon un critère de tri particulier, les mots étant simplement rangés dans leur ordre chronologique de création.

Le dictionnaire, qui regroupe tous les vocabulaires, présente une structure classique de liste chaînée.

Chaque mot est représenté par son nom, une série de pointeurs, et sa définition semi-compilée.

Citons déjà le pointeur de lien, par l'intermédiaire duquel s'effectue la recherche d'un mot dans le vocabulaire. Il contient pour chaque mot l'adresse du mot précédent dans la liste. Pour le premier mot défini, ce pointeur contient la valeur zéro.

Il est également nécessaire de mémoriser l'adresse du dernier mot créé. Toute recherche s'effectue en partant de ce mot vers le premier défini.

Exemple :

Le mot VLIST permet d'obtenir la liste exhaustive des mots du vocabulaire de travail, pour l'instant FORTH. La nature du chaînage fait qu'ils apparaissent dans l'ordre chronologique inverse.

Un des intérêts de FORTH est que le dictionnaire de base est modulaire. L'utilisateur peut commencer à travailler avec un vocabulaire très restreint. Il peut à tout moment, selon ses besoins, faire appel à des modules spécialisés de FORTH.

Les modules existants sur les principaux FORTH sont les suivants :

— travail en double longueur (32 bits)
— calcul flottant
— fonctions mathématiques et graphiques
— gestion de l'imprimante
— gestion de mémoire virtuelle, éditeur
— assembleurs
— gestion des entrées-sorties

Ces découpages permettent à l'utilisateur d'optimiser le vocabulaire en fonction de l'application, ce qui entraîne les avantages suivants :

— réduction de l'encombrement mémoire

— réduction du temps de recherche dans le vocabulaire, et donc du temps d'interprétation

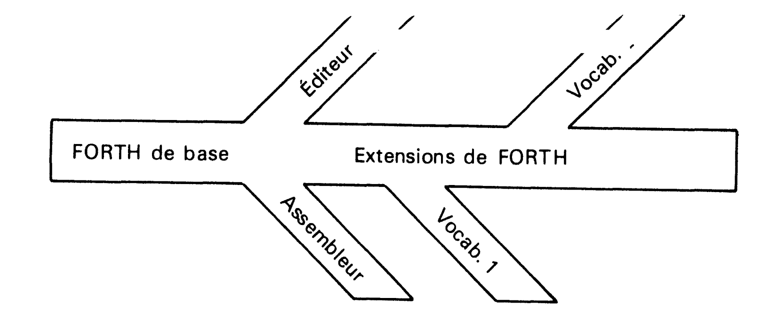
II.4. GÉNÉRALISATION DE LA NOTION DE VOCABULAIRE

Jusqu'à maintenant, nous avons présenté le travail du programmeur comme la définition de nouveaux mots participants du vocabulaire FORTH.

Mais il a de plus la possibilité de créer de nouveaux vocabulaires. Il est en effet possible de travailler dans des vocabulaires de contextes différents, spécialisés pour telle ou telle application.

Cette structure générale de vocabulaires est ramifiée, mais reste transparente à l'utilisateur. La machine ne connaît à un instant donné que le vocabulaire de contexte dans lequel l'utilisateur s'est placé.

Le schéma suivant illustre bien cette structure :



L'ensemble des mots appartenant aux différents vocabulaires constitue le dictionnaire FORTH.

Nous verrons plus tard comment l'utilisateur peut créer ses propres vocabulaires.

Les implantations classiques de FORTH comprennent trois vocabulaires : FORTH, ÉDITEUR, ASSEMBLEUR.

Pour passer d'un vocabulaire à un autre, il suffit de taper le nom du vocabulaire désiré.

Exemple : EDITOR

EDITEUR [Return] OK

FORTH [Return] OK

RESUME DU CHAPITRE II

Mot	*Syntaxe*	*Définition*
FORGET	FORGET "nom"	Élimine du dictionnaire tous les mots définis depuis (chronologiquement) et y compris "nom".
:	: "nom" "définition" ;	Mot qui indique que l'utilisateur définit un nouveau mot. C'est le mot de début de définition.
;	: "nom" "définition" ;	Mot qui indique la fin de définition d'un mot. ; ne peut être utilisé qu'avec : .
(	("commentaire")	Mot qui indique que le texte qui suit est un simple commentaire. Ce mot est toujours utilisé en paire avec) .
)	("commentaire")	Mot qui indique la fin de commentaire.
VARIABLE	"valeur initiale" VARIABLE "nom de la variable"	Mot de définition d'une variable.
CONSTANT	"valeur" CONSTANT "nom de la constante"	Mot de définition d'une constante. La différence fondamentale entre une constante et une variable est que, une fois définie la valeur d'une constante ne peut être modifiée.
.	. (n –––)	Dépile une valeur signée sur 16 bits et l'affiche sur le terminal dans la base courante. La valeur affichée n'est plus sur la pile.
VLIST	VLIST	Affiche sur le terminal la liste des mots FORTH qui constituent le vocabulaire de contexte.
FORTH	FORTH	Rend FORTH le vocabulaire de contexte.
EDITOR	EDITOR	Rend EDITOR le vocabulaire de contexte.
ASSEMBLEUR	ASSEMBLEUR	Rend ASSEMBLEUR le vocabulaire de contexte.

22

Conventions de notation

1. Les touches de contrôle :

— Return ou Enter (suivant les terminaux et les machines)

— Control : Ctl

— Escape : Esc

— Shift

— Break

seront représentées dans un rectangle :

[Return]

[Ctl]

[Esc]

[Shift]

[Break]

EXERCICE 1 :

Que reste-t-il sur la pile à la fin des séquences suivantes :

a) 1 2 3 . . 4 . . 5 .
b) 23 45 . 33 54 1 8 . .
c) 32 15 11 7 . .
d) 9 . 12 . 13 14 . 15 18

EXERCICE 2 :

Transformez les expressions suivantes en notation polonaise inversée :

a) (a+b)*(c+d)
b) ((a+b)/c)+((d+e)*f)
c) (a/(b+c))/((d+e)*(a/c))
d) (a*(b/c))+((d+a)*(b/(c+d)))

23

III

COMMENT PROGRAMMER

III.1. LES OBJETS ET LEUR MANIPULATION

III.1.1. Lire et écrire en mémoire

FORTH travaille sur des cellules de 16 bits. Pour lire le contenu d'une cellule mémoire, il suffit de placer l'adresse de la cellule désirée sur la pile, et d'exécuter le mot @ (Fetch), qui retourne le contenu dans la pile.

Exemple: Lire le contenu de l'adresse 12588:

12588 @ . [Return] 5163 OK

Il existe un mot FORTH qui affiche directement le contenu d'une cellule donnée à l'écran: c'est le mot ? .

Dans notre exemple, on pourra donc aussi taper:

12588 ? [Return] 5163 OK

La coutume est d'exprimer les adresses mémoires en base 16 (hexadécimale). FORTH permet de changer facilement de base, à l'aide des mots HEX (→ 16), et DECIMAL (→ 10).

Exemple:

12588 HEX . [Return] 312C OK

Pour écrire une valeur sur 16 bits dans une cellule mémoire, le principe est le même: placer la valeur puis l'adresse dans la pile, et exécuter le mot ! (Store).

Exemple: Écrire 3F00 à l'adresse 312C:

HEX 3F00 312C ! [Return] OK

Il est possible de travailler sur des cellules de huit bits (1 octet), en utilisant les mots C@ et C! .

Exemple: Relire le contenu des adresses 312C et 312D:

312C C@ . [Return] 0 OK

312D C@ . [Return] 3F OK

On remarque que le mot ! présenté ci-dessus place le LSB (Least Significant Byte) à l'adresse spécifiée, et le MSB (Most Significant Byte) à l'adresse qui suit.

III.1.2. Définir et utiliser des constantes

Nous avons déjà vu comment définir et utiliser pour la première fois une constante:

Valeur CONSTANTE Nom

Dès que la constante a été définie, le nouveau mot FORTH "Nom" a pour action de placer la valeur de la constante sur la pile de donnée.

Ainsi:

16 CONSTANTE SEIZE [Return] OK

SEIZE . [Return] 16 OK

III.1.3. Définir et utiliser des variables

Pour définir une variable, le principe reste le même:

Valeur VARIABLE Nom

Dès que la variable a été définie, le nouveau mot FORTH "nom" a pour action, cette fois, de placer l'adresse de la valeur sur la pile.

Ainsi:

15 VARIABLE LONGUEUR [Return] OK

LONGUEUR [Return] OK (Adresse ---)

@ [Return] OK (Valeur ---)

. [Return] 15 OK (---)

En d'autres termes:

LONGUEUR ? [Return] 15 OK

Pour modifier la valeur d'une variable, on va naturellement utiliser le mot ! déjà présenté.

Exemple:

14 LONGUEUR ! [Return] OK

LONGUEUR ? [Return] 14 OK

III.1.4. Les USER VARIABLES

Les USER VARIABLES sont des variables FORTH spécialisées utilisées par l'interpréteur.

Exemple: — HERE a pour valeur l'adresse du premier emplacement libre dans le dictionnaire.

— BASE a pour valeur la base courante de travail.

A ce propos, la définition de HEX s'écrit:

: HEX 16 BASE ! ; [Return] OK

Pour définir OCTAL, qui nous fera passer dans la base huit, il suffira de faire:

: OCTAL 8 BASE ! ; [Return] OK

Pour connaître la base courante, il suffit de taper:

BASE ? [Return] 10 OK

On peut noter qu'afficher la base courante dans la base courante est un problème singulièrement absurde, puisque la valeur sera toujours 10.

Si maintenant, plus sérieusement, on désire afficher la base courante en décimal, et sans la changer, on pourra définir le mot suivant:

```
: QUELLE-BASE
      BASE @        ( Val-base ––– )
      BASE @        ( Val-base Val-base ––– )
      DECIMAL       ( Val-base Val-base ––– )
      .             ( Val-base ––– )
      BASE          ( Val-base Ad-base ––– )
      !             ( ––– )
; [Return] OK
```

On peut alors exécuter :

```
HEX QUELLE-BASE [Return] 16 OK
```

III.2. LES PILES ET LEUR MANIPULATION

III.2.1. La pile de donnée

Cette pile a déjà été présentée en détail, nous allons maintenant décrire les outils de manipulation qui s'y rattachent.

Lors de la présentation du mot . , nous avons vu que son exécution détruisait systématiquement le sommet de la pile. Comment donc consulter le sommet sans le détruire ? Il suffit de commencer par le dupliquer : c'est le mot DUP.

Exemple :

```
1 [Return] OK          ( 1 ––– )
DUP [Return] OK        ( 1 1 ––– )
. [Return] 1 OK        ( 1 ––– )
```

Il existe quatre autres mots importants :

— Le mot DROP (n –––), qui détruit le haut de la pile.

— Le mot SWAP (n1 n2 ––– n2 n1), qui échange les deux dernières données entrées.

— Le mot OVER (n1 n2 ––– n1 n2 n1), qui duplique l'avant dernière donnée entrée sur le sommet de la pile.

— Le mot ROT (n1 n2 n3 ––– n2 n3 n1), qui effectue une permutation circulaire sur les trois dernières données.

Toutes ces manipulations concernent des données sur 16 bits, utilisant une seule cellule de la pile.

Cependant, ces mots ont chacun leur équivalent pour la manipulation de doubles cellules (32 bits) :

```
2DUP      ( n1 n2 ––– n1 n2 n1 n2 )
2DROP     ( n1 n2 ––– )
2SWAP     ( n1 n2 n3 n4 ––– n3 n4 n1 n2 )
2OVER     ( n1 n2 n3 n4 ––– n1 n2 n3 n4 n1 n2 )
2 ROT     ( n1 n2 n3 n4 n5 n6 ––– n3 n4 n5 n6 n1 n2 )
```

On remarque que la plupart de ces mots a une action différente de celle de la répétition des mots correspondants travaillant sur 16 bits.

Exemple :

```
DUP DUP   ( n1 n2 ––– n1 n2 n2 n2 )
2DUP      ( n1 n2 ––– n1 n2 n1 n2 )
```

III.2.2. La pile de RETURN

Comme tous les langages évolués, FORTH utilise une pile LIFO spécialisée pour mémoriser les adresses de retour aux procédures appelantes.

La différence essentielle est que FORTH donne à l'utilisateur l'accès à cette pile grâce aux deux mots suivants :

>R	Transfère la cellule du sommet de la pile des données vers le sommet de la pile de return.
R>	Effectue l'opération inverse.

L'utilisation la plus courante de cette pile auxiliaire réside dans les déchargements provisoires de la pile de données.

Par exemple, on peut donner facilement la définition des mots 2DUP, 2DROP, 2SWAP, 2OVER et 2ROT, en utilisant cette pile de return :

```
: 2DUP     OVER OVER ;  [Return] OK
: 2DROP    DROP DROP ;  [Return] OK
: 2SWAP    ROT >R ROT R> ;  [Return] OK
: 2OVER    >R >R 2DUP R> R> 2SWAP ;  [Return] OK
: 2ROT     >R >R 2SWAP R> R> 2SWAP ;  [Return] OK
```

Il existe d'autres utilisations plus rares de cette pile. Par exemple, tapez :

```
: BONSOIR 123 >R ;  [Return] OK
BONSOIR  [Return]
```

Vous avez compris, si vous avez fait l'essai, que ce type d'utilisation est réservée à une élite de barbus.

En effet, il ne faut jamais oublier que l'interpréteur utilise en permanence cette pile, pour y mémoriser les adresses de retour des procédures. Si vous l'utilisez en cours de définition, vous êtes priés de la laisser en fin de définition dans l'état où vous auriez aimé la trouver en début de définition.

III.3. LES MOTS ARITHMÉTIQUES

III.3.1. Présentation

Le FORTH dispose de mots arithmétiques spécialisés pour chacun des types de données suivant :

— les nombres en simple longueur (16 bits) signés,
— les nombres en simple longueur non signés,
— les nombres en double longueur (32 bits) signés,
— les nombres en double longueur non signés,
— les nombres flottants.

Il existe de plus des mots de conversion d'un type à l'autre, ainsi que des mots dits mixtes, travaillant sur plusieurs types à la fois.

Il est d'usage de représenter chacun de ces types sur la pile de la manière suivante :

n : simple longueur signée
u : simple longueur non signée
d : double longueur signée
ud : double longueur non signée
f : flottant

Les nombres négatifs sont représentés de façon très classique par leur complément à deux. Le principe de cette représentation consiste à compléter tous les bits du nombre à 1, puis d'ajouter 1 au nombre ainsi obtenu.

Exemple :

```
12 est représenté par      :  0000 0000 0000 1100
soit en hexadécimal        :   0    0    0    C
—12 sera représenté par       1111 1111 1111 0011
                             +                  1
                              -------------------
                              1111 1111 1111 0100
soit en hexadécimal        :   F    F    F    4
```

Il faut bien avoir en tête que la valeur du nombre contenu dans une cellule dépend du type que l'on lui alloue. En effet, supposons qu'une cellule contienne le nombre hexadécimal FFF4. Si l'on considère celui-ci comme signé, sa valeur sera —12. Si on le considère comme nombre non signé, sa valeur sera 65524.

III.3.2. Les mots simple longueur

FORTH dispose bien sûr des quatre mots de base, + — * / , qui opèrent ici sur des nombres signés. Le mot / représente ici la division entière.

Exemple :

```
13 5 / .  [Return] 2 OK
```

L'effet de ces mots sur la pile se représente ainsi :

```
+    ( n1 n2 --- n1+n2 )
—    ( n1 n2 --- n1—n2 )
*    ( n1 n2 --- n1*n2 )
/    ( n1 n2 --- n1/n2 )
```

Pour obtenir le reste de la division entière, on utilisera le mot MOD, travaillant sur des nombres non signés.

Exemple :

```
13 5 MOD .  [Return] 3 OK
MOD   ( u1 u2 --- u—reste )
```

Il existe un mot qui fournit à la fois le quotient et le reste de la division entière, c'est le mot /MOD.

Exemple:

```
13 5 /MOD . . [Return] 2 3 OK
/MOD ( u1 u2 --- u-reste u-quotient )
```

Vous pouvez vérifier que /MOD pourrait se définir à l'aide des mots MOD et / de la manière suivante:

```
: /MOD 2DUP MOD ROT ROT / ; [Return]    OK
```

Supposons maintenant que vous ayez à calculer un pourcentage, par exemple 45 % de 11241. Vous pourriez être tenté d'utiliser une des deux solutions qui suivent:

```
    11231 45 * 100 /
ou  11241 100 / 45 *
```

La première solution serait rejetée par la machine, le produit 45*11241 ne tenant pas sur 16 bits.

Vous seriez donc contraints d'utiliser la seconde. Cependant, la division entière vous ferait perdre beaucoup de précision.

Pour pallier facilement à ces inconvénients, il existe un mot FORTH qui enchaîne les opérations * et /, et qui utilise un résultat intermédiaire sur 32 bits: le mot */.

```
*/      ( n1 n2 n3 --- n1*n2/n3 )
```

Vous ferez alors:

```
11241 45 100 */
```

La division reste ici la division entière, et il existe donc un mot qui fournit à la fois le quotient et le reste, tout en respectant les contraintes de */: le mot */MOD.

```
*/MOD   ( u1 u2 u3 --- u-reste u-quotient )
```

Exemple:

```
11241 45 100 */MOD . .  [Return] 5058 45 OK
```

Voici maintenant quatre mots d'usage plus classique:

```
ABS      ( n --- |n| )
NEGATE   ( n --- -n )
MIN      ( n1 n2 --- Min(n1,n2) )
MAX      ( n1 n2 --- Max(n1,n2) )
```


Il existe une autre famille de mots arithmétiques, dont l'un des opérandes est fixé. Par exemple 1+, 2+, 1−, 2*, etc...

Leur utilisation est évidente:

Exemple:

```
5 2* .  [Return] 10 OK
```

Leur intérêt est uniquement relatif à l'optimisation du temps d'interprétation. En effet, il est courant d'avoir à faire au cours d'une définition, des opérations du type ajouter 1, soustraire 2, etc... L'utilisation du mot 1+, par exemple, plutôt que de la syntaxe 1 +, ne nécessite qu'une seule recherche de la part de l'interpréteur, et donc accélère l'interprétation.

Les mots de ce type les plus souvent définis sont les suivants:

```
1+    ( n --- n+1 )
1-    ( n --- n-1 )
2+    ( n --- n+2 )
2-    ( n --- n-2 )
2*    ( n --- n*2 )
2/    ( n --- n/2 )
```

Pour conclure, citons le mot U. qui, contrairement au mot . qui affiche le sommet de la pile comme une valeur signée, affiche cette même valeur non signée.

Exemple:

```
-12 .   [Return] -12 OK
-12 U.  [Return] 65524 OK
```

III.3.3. Les mots double longueur

L'addition et la soustraction en double longueur s'effectuent par l'intermédiaire des mots D+ et D−:

```
D+      ( d1 d2 --- d1+d2 )
D-      ( d1 d2 --- d1-d2 )
```

De même, on dispose des équivalents de NEGATE, ABS, MIN et MAX:

DNEGATE (d --- -d)

DABS (d --- |d|)

DMIN (d1 d2 --- Min(d1,d2))

DMAX (d1 d2 --- Max(d1,d2))

Il est nécessaire de disposer de mots permettant des entrées-sorties sur la pile en double longueur.

Pour introduire un nombre en double longueur dans la pile, il suffit de faire précéder immédiatement son expression d'un délimiteur particulier, le plus souvent le point. L'empilage a toujours lieu dans le sens LSW, MSW.

Exemple :

.12 [Return] OK

. [Return] 0 OK

. [Return] 12 OK

Pour afficher directement un nombre en double longueur situé au sommet de la pile, on utilise le mot D. :

D. (d ---)

Exemple :

.12 D. [Return] 12 OK

12 0 D. [Return] 12 OK

0 2 D. [Return] 131072 OK

Il existe un mot de conversion de type, qui convertit un nombre en simple longueur signé placé en haut de la pile, en un nombre en double longueur signé :

S->D (n --- d)

Exemple :

12 S->D D. [Return] 12 OK

III.3.4. Les mots mixtes

Il peut être pratique de pouvoir par exemple additionner les nombres de types différents, sans avoir à faire de conversion. Dans ce domaine, FORTH dispose des mots suivants :

M+ (d n --- d-somme)

M/ (d n --- n-quotient) (division entière)

U/MOD (ud u --- u-reste u-quotient)

M* (n1 n2 --- d-produit)

U* (u1 u2 --- ud-produit)

M*/ (d n u --- d-résultat)

De même que */ utilisait un résultat intermédiaire en double longueur, M*/ utilise un résultat intermédiaire en triple longueur.

Nous vous recommandons de vous familiariser avec ces différents mots en les utilisant sur des exemples de votre choix.

III.3.5. Controverse point flottant, point fixe

Comme vous l'avez sans doute remarqué, nous n'avons jusqu'alors présenté que des mots arithmétiques travaillant sur des nombres entiers.

— Principe du point fixe :

Dans ce système de stockage des nombres, c'est le programme qui prend en charge la gestion de la virgule, qui n'est pas connue de la machine.

Par exemple, si vous devez faire des calculs de gestion comptable, vous pourrez parfaitement exprimer tous les nombres en centimes, faire tous les calculs sur des valeurs alors entières (simple ou double longueur), puis ne faire apparaître la virgule décimale qu'au moment de l'édition des résultats, si vous désirez que ces derniers soient exprimés en francs.

Les valeurs numériques sont stockées dans des emplacements de longueur fixe, et au cours des calculs il y a donc risque de dépassement de capacité.

En double longueur, la gamme des nombres représentables s'étend donc de − 2 147 483 648 à + 2 147 483 647.

Un des travaux préliminaires du programmeur est donc d'exprimer, en fonction de la précision choisie, l'échelle optimale de représentation.

Ainsi, en gestion financière, on convertira tous les nombres en kilofrancs.

Cette démarche est sur tous les plans la plus naturelle. Voici comment elle s'inscrit parfaitement dans quelques exemples classiques d'applications :

— *A[illegible]ition [illegible]don[illegible] tem[illegible]el*

Si vous avez déjà examiné un appareil de mesure moderne, disposant d'un système d'affichage, vous aurez remarqué que les données sont calibrées par un facteur d'échelle au gré de l'expérimentateur, et apparaissent en format fixe. Le point décimal est même parfois gravé sur le panneau avant lui-même: c'est typiquement la philosophie de la virgule fixe.

Passons au cas plus général de l'acquisition de données toujours présente dans un problème de temps réel.

L'organe intermédiaire entre un processus physique et un calculateur est composé d'un ou de plusieurs capteurs suivis par une amplification et une conversion analogique-numérique. L'ensemble électronique amplificateur-convertisseur fournit par son choix même la gamme de valeurs qui apparaîtront. Le choix de l'échelle pour les calculs en virgule fixe est donc immédiatement induit par cet organe d'interface.

— *Gestion*

Une des phases de l'analyse d'une application de gestion classique, consiste en une dissection en profondeur de toutes les données. La longueur des informations alpha-numériques est fixée, de même que les valeurs maximales et minimales de toutes les données numériques.

C'est le travail préparatoire à un calcul en virgule fixe.

Le travail en virgule fixe, qui est très souvent possible, est le meilleur choix car il permet, pour des raisons purement techniques que nous allons évoquer par la suite, de diviser les temps de calculs par un coefficient supérieur à trois.

— Principe du point flottant:

Pour un programmeur utilisant maladroitement le point fixe, le problème de dépassement de capacité se présente trop souvent. Son souhait serait de pouvoir confier à la machine la gestion automatique des changements d'échelle, par exemple dans les multiplications.

Cette possibilité lui est offerte par le travail en point flottant.

Les données sont exprimées dans la machine sous la forme:

Signe Exposant Mantisse

ou l'exposant est calculé de telle sorte que la mantisse soit cadrée à gauche.

Cette prise en charge automatique des changements d'échelle par la machine a certainement été trop galvaudée, et a développé une tendance à la paresse dans la programmation.

Quels sont les emplois légitimes du point flottant:

1. Vous voulez utiliser une machine comme calculateur en point flottant.

2. Vous valorisez le temps de développement par rapport au temps d'exécution.

3. La gamme de variation de vos données est vraiment trop grande.

4. Vous disposez d'un processeur cablé flottant qui optimise ce type de calcul.

FORTH pour sa part a beaucoup plus d'affinités pour le point fixe que pour le point flottant, car il cadre mieux avec le caractère sobre et précis du programmeur FORTH.

Il faut bien garder en tête qu'un calcul en flottant, aussi confortable qu'il soit pour le programmeur, demande à la machine un beaucoup plus grand nombre d'opérations que son équivalent en point fixe. Les temps d'exécution en sont les premières victimes.

Comparons par exemple une multiplication traitée des deux manières.

En point fixe, cette dernière se résumera à l'algorithme classique d'addition-décalage.

En point flottant, la machine devra d'une part additionner les exposants, puis effectuer la multiplication des mantisses, puis recalculer l'exposant de manière à cadrer la mantisse à gauche.

L'occupation de la mémoire pendant le calcul n'est donc pas non plus optimisé, puisqu'il faut traiter parallèlement les exposants et les mantisses.

En point fixe, la machine ne connaît que des entiers avec lesquels elle calcule très rapidement.

Cette tendance du FORTH pour le point fixe est tellement marquée que les premières versions du FORTH ne connaissaient pas les calculs en flottant, qui furent introduits par la suite sous la forme de modules comme décrit précédemment.

III.4.1. Les variables booléennes

Comme dans de nombreux langages évolués, les variables booléennes ne sont pas physiquement différentes des autres variables. L'état FAUX correspond à une valeur nulle, toutes les autres valeurs correspondant à un état VRAI.

Le principe de fonctionnement des mots de comparaison est le suivant : les paramètres sont rangés au préalable dans la pile, et le résultat de la comparaison sera retourné en simple longueur dans la pile sous la forme d'une variable booléenne.

(var1 var2 --- flag)

On conviendra par la suite de représenter les paramètres booléens sur la pile par le symbole b.

III.4.2. Les comparaisons simple longueur

FORTH connaît la liste des mots de comparaison suivants :

=	(n1 n2 --- b)	teste si n1=n2
U<	(u1 u2 --- b)	teste si u1<u2
<	(n1 n2 --- b)	teste si n1<n2
>	(n1 n2 --- b)	teste si n1>n2
0=	(n --- b)	teste si n=0
0>	(n --- b)	teste si n>0
0<	(n --- b)	teste si n<0

NOTA : Pour pouvoir effectuer des comparaisons de nombres supérieurs à 32767, il faudra le spécifier à l'interpréteur, afin qu'il ne les considère pas comme signés. Pour bien percevoir cette notion, faites l'essai suivant :

6000 A000 < . [Return] 0 OK

6000 A000 U< . [Return] 1 OK

Méditez ...

III.4.3. La comparaison en double longueur

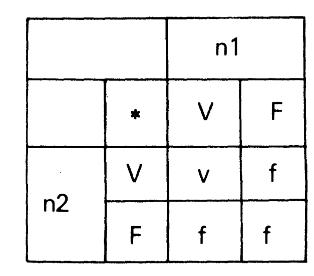
Les mots appartenant à cette catégorie sont les suivants :

D=	(d1 d2 --- b)	teste si (d1=d2)
D<	(d1 d2 ---b)	teste si (d1<d2)
DU<	(ud1 ud2 --- b)	teste si (ud1<ud2) sur 32 bits

III.4.4. Les opérateurs booléens

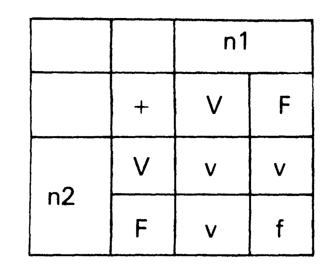
Les opérateurs booléens traditionnellement employés sont : ET, OU, complémentation, représentés en FORTH respectivement par *, +, 0=.

Pour mémoire, les tables de vérité de ces 3 opérateurs logiques sont :



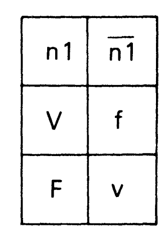
		n1	
	*	V	F
n2	V	v	f
	F	f	f

ET logique



		n1	
	+	V	F
n2	V	v	v
	F	v	f

OU logique



n1	$\overline{n1}$
V	f
F	v

Complémentation

Exemple d'application : Tester si un nombre appartient à un intervalle ouvert.

Les valeurs seront entrées dans la pile dans l'ordre :

(Borne–inf n Borne sup ---)

Le résultat du test étant lui-même retourné dans la pile.

Le test s'écrit de façon très simple :

(n > Borne–inf) ET (n < Borne-sup)

et devient en FORTH :

```
...ER\   :          (    ıf n     ıp —
            OVER    ( B—Inf n B—Sup n ——— )
            >       ( B—Inf n b1 ——— )
            >R      ( B—Inf n ——— )
            <       ( b2 ——— )
            R>      ( b2 b1 ——— )
            *       ( b2 et b1 ——— )
; [Return] OK
```

1 2 3 INTERVALLE . [Return] 1 OK

Certains langages connaissent le OU-EXCLUSIF dont la table de vérité est :

	n1		
n2	OU—EX	V	F
	V	f	v
	F	v	f

A titre d'exercice, définissons le mot OUEX qui remplit cette fonction.

On voit que A OUEX B s'écrit de façon très simple :

(A OU B) ET ($\overline{A}$ OU $\overline{B}$)

```
: OUEX            ( n1 n2 ——— )
        2DUP      ( n1 n2 n1 n2 ——— )
        +         ( n1 n2 (n1 ou n2) ——— )
        >R        ( n1 n2 ——— )
        0=        ( n1 n̄2 ——— )
        SWAP      ( n̄2 n1 ——— )
        0=        ( n̄2 n̄1 ——— )
        +         ( (n̄1 ou n̄2) ——— )
        R>        ( (n̄2 ou n̄1) (n1 ou n2) ——— )
        *         ( n1 OUEX n2 ——— )
; [Return] OK
```

1 3 OUEX . [Return] 0 OK

0 15 OUEX . [Return] 15 OK

FORTH connaît également des mots de manipulation de bits, dont les mnémoniques peuvent prêter à confusion, puisqu'il s'agit de AND, NOT, OR, XOR.

Ces mots effectuent respectivement les opérations ET, OU, COMPLÉMENTATION, OU—EX, mais en parallèle sur chaque bit.

Ainsi :

1 2 AND . [Return] 0 OK

En effet,

1 —> 0000 0001

2 —> 0000 0010

1 AND 2 —> 0000 0000

Alors que :

1 2 + . [Return] 3 OK

III.5. LES MOTS DE STRUCTURATION

L'utilisateur devra particulièrement s'intéresser à ces mots, car de leur emploi à bon escient dépendra la qualité du logiciel.

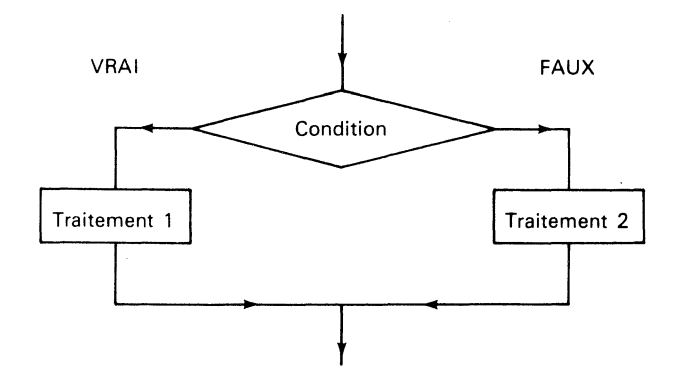
En effet, au cours du développement, leur élégante utilisation sera l'aboutissement d'une bonne analyse, d'idées claires et précises ainsi que d'une compréhension globale du problème.

Ils permettront à l'utilisateur d'écrire un programme souple, lisible, bien équilibré, caractéristiques extrêmement appréciables lorsque l'on veut fournir des logiciels à longue durée de vie.

On y retrouve les structures conditionnelles et répétitives classiques, chacun des mots nécessitant des paramètres les prenant sur la pile comme à l'accoutumée.

III.5.1. La structure conditionnelle IF ... ELSE ... THEN

L'organigramme d'une telle structure est le suivant :



Rappelons que, dans l'esprit FORTH, tout paramètre utilisé par un mot doit être d'abord préparé éventuellement sur la pile. Ceci risque de troubler quelques lecteurs. En effet, une syntaxe classique : IF conditions ... aura pour équivalent en FORTH : conditions IF ...

La syntaxe FORTH correspondante sera :

Condition IF traitement 1 ELSE traitement 2 THEN.

Seule la valeur du booléen présent sur la pile au moment de l'exécution du IF détermine le choix du traitement :

```
b = VRAI  traitement 1
b = FAUX  traitement 2
```

L'effet de ces mots sur la pile s'écrit :

```
IF      ( b --- )
ELSE    ( --- )
THEN    ( --- )
```

Exemple : Notre but est de recréer notre mot INTERVAL. Nous pouvons maintenant nous permettre de n'effectuer le second test (N>Binf) que si le premier (N<Bsup) s'est avéré être vrai :

```
: INTERVAL        (Binf n Bsup ---)
     OVER         (Binf n Bsup n ---)
     >            (Binf n b1 ---)
     IF
         <        (b2 ---)
     ELSE
         2DROP
         0
     THEN
; Return OK
```


On peut aisément utiliser plusieurs de ces structures en les imbriquant.

Pour notre exemple, nous présentons ici un mot FORTH permettant d'afficher une chaîne de caractères sur le périphérique de sortie, le mot ." .

La chaîne de caractères doit être située après ce mot, et se terminera par le caractère " .

Exemple :

```
: BONJOUR ." CA VA ?" ; Return OK
BONJOUR   Return CA VA? OK
```

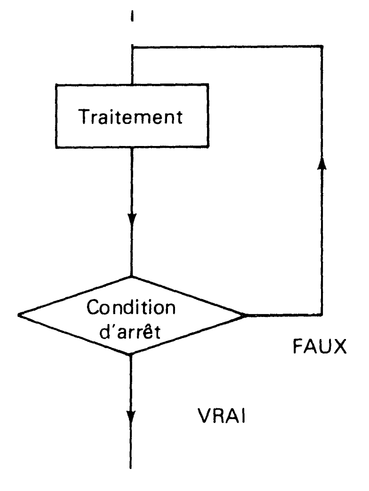
Nous voulons maintenant que notre mot INTERVAL ne nous retourne plus un boléen sur la pile, mais nous affiche l'une des trois réponses suivantes : TROP PETIT, BON, TROP GRAND ; On écrira :

```
: INTERVAL
     OVER
     >
     IF
        <
        IF
           . "BON"
        ELSE
           . "TROP PETIT"
        THEN
     ELSE
        2DROP
        . "TROP GRAND"
     THEN
; Return OK
```

III.5.2. Les structures répétitives

a) La structure DO ... LOOP, boucle définie.

L'algorithme est le suivant :



L'action des mots sur la pile de données s'écrit:

```
DO        ( Ind(Fin)+1 Ind(Début) ––– )
LOOP      ( ––– )
```

Ces deux mots effectuent les actions suivantes:

DO empile sur la pile de return l'indice de fin + 1, ainsi que l'indice de début, puis enchaîne l'exécution des mots du traitement à répéter.

LOOP compare les deux valeurs au sommet de la pile de return. Si le dernier est inférieur à l'avant dernier, LOOP incrémente le sommet de 1 et transfère l'exécution au mot qui suit immédiatement DO. Dans le cas contraire, la pile des return est nettoyée de ses deux valeurs au sommet, et l'exécution est passée au premier mot suivant LOOP.

Notons tout de suite que le traitement répétitif s'exécutera au moins une fois.

Exemple: Afficher tous les nombres de 1 à 10:

```
: AFFICHE  1        ( 1 ––– )
           10 0     ( 1 10 0 ––– )
           DO       ( 1 ––– )
              DUP   ( 1 1 ––– )
              .     ( 1 ––– )
              1+    ( 2 ––– )
           LOOP
           DROP     ( ––– )
           ; Return  OK
AFFICHE Return 1 2 3 4 5 6 7 8 9 10 OK
```


Voici maintenant un mot d'utilisation pratique dans le cadre de cette structure: le mot I. Celui-ci duplique sur la pile des données le sommet de la pile de return, c'est-à-dire l'indice courant lors de l'exécution d'un DO ... LOOP.

Exemple: On peut donner une autre définition de AFFICHE.

```
: AFFICHE
        11  1  ( 11 1 ––– )
        DO     ( ––– )
         I  .  ( ––– )
        LOOP   ( ––– )
;Return OK
```

On peut aisément imbriquer les boucles. Supposons que l'on veuille afficher un carré d'étoiles de 3 colonnes et 3 lignes, on écrira:

```
: CARRE
     CR
     3  0            ( indices boucle lignes )
     DO
        3  0         ( indices boucle colonnes )
        DO
          ." *"
        LOOP
        CR
     LOOP
;Return OK
```

Le mot CR émet le caractère ASCII de code 13 (carriage return) sur le périphérique de sortie, et fait donc passer à la ligne.

```
CARRE  Return
***
***
***
OK
```

De même que le mot I duplique l'indice de la boucle en cours d'exécution, il exite un mot K qui duplique sur la pile de données l'indice de la boucle de niveau immédiatement supérieur, c'est-à-dire la troisième cellule de la pile de return.

On peut imaginer une manière artificielle de sortir d'une boucle qui consiste à remplacer le sommet de la pile de return par le contenu de la

deux cell Un ı qui r it ce ... nction est le mot LEAVE. On pourrait le définir comme suit :

```
: LEAVE
        R> R>    ( cel1 ––– )
        DROP     ( ––– )
        R>       ( cel2 ––– )
        DUP      ( cel2 cel2 ––– )
        >R >R >R
; [Return] OK
```

Précisons cependant que ce mot n'est utilisé que pour se sortir de situations inextricables amenées par une mauvaise préparation, et un mauvais choix de la structure de contrôle. Son utilisation n'est donc pas à conseiller.

Si le programmeur désire incrémenter son indice courant d'une valeur différente de 1, il pourra utiliser la syntaxe :

```
DO ...+LOOP          +LOOP    ( Incr ––– )
```

L'incrément devra alors se situer sur la pile de données au moment de l'exécution de +LOOP. Le fonctionnement général de cette structure est absolument identique à celui de DO...LOOP.

Exemple :

```
: TABLE          ( n ––– )
    DUP          ( n n ––– )
    10 * 1+      ( n 10n+1 ––– )
    OVER         ( n 10n+1 n ––– )
    DO
         I .     ( n ––– )
         DUP
    +LOOP
    DROP
;[Return]OK
5 TABLE [Return] 5 10 15 20 25 30 35 40 45 50 OK
```

b) La structure BEGIN ... AGAIN, boucle infinie.

L'algorithme est le suivant :



Actions sur la pile :

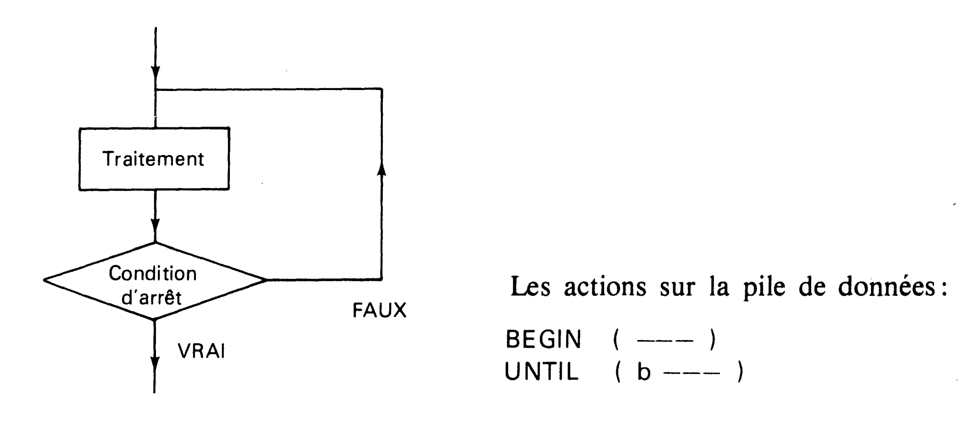
```
BEGIN   ( ––– )
AGAIN   ( ––– )
```

Cette structure présente relativement peu d'intérêt puisqu'il n'existe pas de moyens de la quitter en douceur et que la boucle infinie reste encore la honte du programmeur.

Cependant, elle peut trouver emploi dans des systèmes de contrôle de processus, systèmes d'exploitation, etc... Nous vous laissons toute liberté de vous lancer dans un exemple de ce type.

c) La structure BEGIN ... UNTIL, boucle indéfinie.

L'algorithme est le suivant :



Les actions sur la pile de données :

```
BEGIN   ( ––– )
UNTIL   ( b ––– )
```

Le débranchement se fait ici au niveau du UNTIL, uniquement si le booléen au sommet de la pile a alors une valeur VRAI. Dans le cas contraire, l'exécution est transférée au mot suivant BEGIN.

On remarque que le traitement est effectué ici aussi au moins une fois.

Supposons pour notre exemple que nous disposions des deux mots FORTH, OUVRE et FERME dont l'action soit respectivement l'ouverture et la fermeture d'un robinet.

Supposons d'autre part que l'exécution d'un mot PLEIN ? retourne sur la pile un booléen ayant la valeur VRAI si le lavabo est rempli.

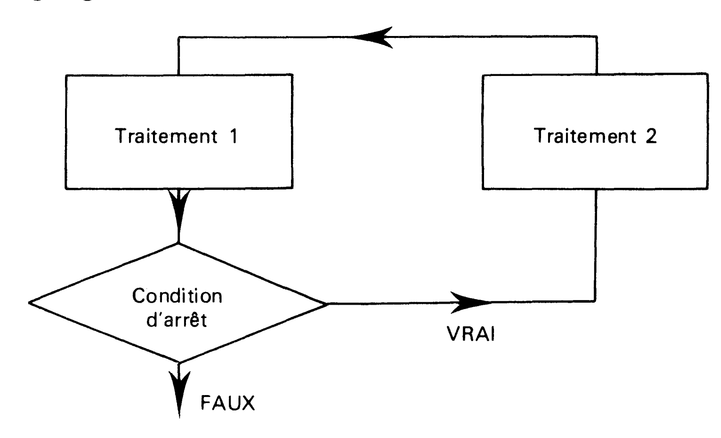
… peut … dé… le m… EM… de … mani… suiva…

```
:REMPLIR OUVRE BEGIN PLEIN? UNTIL FERME; [Return] OK
```

Cette structure peut très souvent remplacer avantageusement une mauvaise utilisation de DO ... LOOP, dans des problèmes où l'introduction d'indices de fin est purement artificielle, et où la condition de sortie survient souvent avant la coïncidence des indices.

d) La structure BEGIN ... WHILE ... REPEAT;

L'organigramme se décrit:



Effet des mots sur la pile:

```
BEGIN    ( ––– )
WHILE    ( b––– )
REPEAT   ( ––– )
```

Cette structure est de loin la plus simple, puisqu'elle permet de placer le point de débranchement à un endroit quelconque dans la boucle.

En effet, celui-ci se situe au niveau du WHILE.

Il y aura débranchement si le booléen alors présent sur la pile prend la valeur FAUX.

Il est de plus important d'avoir en tête que dans l'utilisation de cette structure, il ne suffit pas de mettre 0 sur la pile si l'on veut sortir, il faut aussi y mettre une valeur non nulle si l'on veut y rester !!! Cette remarque est d'ailleurs aussi valable pour la structure BEGIN ... UNTIL, au sens du booléen près.

Pour notre exemple, supposons que l'on dispose d'un mot COULE qui ouvre un robinet pendant un très court instant. On pourra définir REMPLIR ainsi:


```
: F …LIR
        BEGIN
              PLEIN?  ( b ––– )
              0=      ( b ––– )
        WHILE
              COULE
        REPEAT
; [Return] OK
```

Toutes les structures que nous venons de voir sont imbricables, la seule limite étant la taille mémoire de la machine !!!

Nous aurons d'ailleurs l'occasion de les voir souvent appliquées dans les problèmes figurant en fin de l'ouvrage.

III.6. LES ENTRÉES—SORTIES

III.6.1. Le clavier

FORTH dispose comme tous les langages évolués de la possibilité d'interroger le clavier. Il peut le faire aussi bien au niveau du caractère que de la chaîne complète.

Le mot qui travaille sur un caractère est le mot KEY. L'exécution de ce dernier met l'interpréteur en attente d'une frappe au clavier, et retourne sur la pile, lorsqu'une touche est pressée, le code ASCII du caractère correspondant:

```
KEY      ( ––– C )
```

Exemple: Définissons un mot qui attendra la frappe d'une touche, affichera le code du caractère correspondant et répétera cette opération jusqu'à ce que vous frappiez la touche RETURN. Rappelons que cette dernière touche a pour code ASCII 13.

```
^SCII
    BEGIN
        KEY     ( c --- )
        DUP     ( c c --- )
        13 -    ( c c-13 --- )
    WHILE
        .       ( --- )
    REPEAT
    DROP        ( --- )
;Return OK
```

Remarque importante: les nombres figurant dans une définition sont convertis en binaire par rapport à la base courante. Ici, RETURN a pour code ASCII 13 en base décimale. Si vous tapez cette définition sous la base hexadécimale, il faudra remplacer 13 par 0D.

La définition étant semi-compilée, une modification ultérieure de la base courante n'aura bien entendu aucun effet sur les valeurs contenues dans la définition.

Un deuxième mot bloque l'exécution dans l'attente d'une frappe au clavier : il s'agit de EXPECT. Cependant, celui-ci sait attendre la frappe d'une chaîne de caractères complète dont on lui précise la longueur, et la range où l'on veut.

```
EXPECT        ( Adr u --- )
```

L'adresse correspond au premier octet en mémoire à partir duquel on désire ranger la chaîne de caractères. Le nombre non signé correspond à la longueur limite de cette chaîne, l'interpréteur reprenant la main lorsque cette limite est atteinte ou lorsque l'on frappe la touche RETURN.

La définition de EXPECT à partir de KEY pourrait s'écrire :


```
PECT                    ( adr u ---            )
    0                   ( adr u 0 ---          )
    BEGIN
        2DUP            ( adr u l u l ---      )
        -               ( adr u l u-l ---      )
        IF
          ROT           ( u l adr ---          )
          KEY           ( u l adr c ---        )
          DUP           ( u l adr c c ---      )
          13 -          ( u l adr c c-13 ---)
          IF
            DUP EMIT
            OVER        ( u l adr c adr --- )
            C !
            1+          ( u l adr+1 ---        )
            ROT ROT     ( adr+1 u l ---        )
            1+          ( adr+1 u l+1 ---      )
            0           ( adr+1 u l+1 0 ---    )
          ELSE
            2DROP DROP  ( ---                  )
            1           ( 1 ---                )
          THEN
        ELSE
          2DROP  2DROP( ---                    )
          1             ( 1 ---                )
        THEN
    UNTIL
;Return OK
```

Nous allons maintenant nous munir d'un outil qui nous sera précieux par la suite : le mot DUMP, qui affichera à l'écran une zone mémoire :

```
DUMP          ( Adr n --- )
```

L'adresse correspond au début de la zone mémoire à afficher, et le nombre n au nombre d'octets.

On pourrait définir ce mot ainsi :

```
: D[illegible]
        0           ( adr N ind–deb ––– )
        DO
          DUP       ( adr adr ––– )
          I
          +         ( adr adr+I ––– )
          C@        ( adr M ––– )
          .         ( adr ––– )
        LOOP
        DROP        ( ––– )
; Return OK
```

On peut maintenant faire l'essai suivant :

```
HEX Return OK
C000 10 EXPECT Return ABCDEFGHIJKLMNOP OK
C000 10 DUMP Return 41 42 43 44 45 46 47 ...... OK
```

On retrouve bien les caractères tapés sous la forme codée.

Une fois la ligne saisie par l'intermédiaire des deux mots précédemment vus, il faut la mettre en forme pour préparer le travail de l'interpréteur.

La première chose à faire est d'isoler les mots. Le mot WORD effectue ce travail en allant ranger le mot suivant dans la phrase à la fin du dictionnaire (HERE). Le mot en mémoire est précédé par un octet contenant sa longueur. Le séparateur est au choix de l'utilisateur, le plus souvent il s'agira du caractère blanc. L'adresse retournée est le début du mot en mémoire.

```
WORD        ( c ––– ad )
```

Pour bien comprendre l'action de WORD, définissons le mot suivant :

```
: ESSAI
      32 WORD     ( adr ––– )   Met le mot suivant dans la phrase
                                à la fin du dictionnaire.
      6 DUMP      ( ––– )       Affiche les six premiers octets
                                de la fin du dictionnaire.
; Return OK
ESSAI ABCD Return 4 65 66 67 68 72 OK
```

Sur certains FORTH, WORD ne retourne pas d'adresse sur la pile, puisqu'il est bien évident que celle-ci correspond toujours au contenu de la variable – utilisateur HERE.

I[illegible] dra [illegible] dé[illegible] alors ESSAI [illegible] la manière suivante :

```
: ESSAI 32 WORD HERE 6 DUMP; Return OK
```

Le problème de WORD est qu'il transfère le mot à la fin du dictionnaire, dans une zone mémoire donc éminemment variable, car toute nouvelle définition vient s'y installer. Quand vous saurez de plus que l'interpréteur lui-même utilise WORD et donc cette même zone mémoire, vous comprendrez qu'il vaut mieux que notre mot ne s'y attarde pas trop longtemps.

Pour pallier à ce problème, FORTH dispose d'une zone tampon, dont l'adresse est en permanence contenue dans la variable – utilisateur PAD.

Il existe donc un mot équivalent à WORD, qui transfère le mot suivant de la phrase, non plus en fin du dictionnaire, mais dans cette zone tampon : c'est le mot TEXT.

```
TEXT        ( c ––– )
```

c correspond encore ici au code ASCII du caractère délimiteur. Mais ici, la chaîne en mémoire n'est pas précédée de sa longueur. Parallèlement à l'essai de WORD, on peut faire l'essai suivant :

```
: ESSAI0
        32 TEXT
        PAD
        6 DUMP
; Return OK
ESSAI0 ABCD Return 65 66 67 68 32 32 OK
```

Une fois que le mot a été transféré en mémoire par WORD on a souvent besoin d'obtenir la longueur du mot, ainsi que l'adresse où il commence vraiment. L'adresse retournée par WORD est celle de l'octet précédant le mot, qui contient sa longueur.

FORTH dispose d'un mot effectuant cette tâche :

```
COUNT       ( adr ––– adr+1 u )
```

On peut le définir facilement :

```
: COUNT                   ( adr ––– )
      DUP                 ( adr adr ––– )
      C@                  ( adr u ––– )
      SWAP 1+  SWAP       ( adr+1 u ––– )
; Return OK
```

Pour ceux d'entre vous qui ne possèdent pas le mot TEXT dans leur FORTH de base, ils pourront le définir de la manière suivante :

```
: TEXT
        PAD
        72 32 FILL
        WORD
        HERE
        COUNT
        PAD SWAP
        <CMOVE
; Return OK
```

III.6.2. L'écran : problème simple d'affichage

Nous avons déjà utilisé les mots de base . , D. et U. qui servent respectivement à l'affichage de la valeur du sommet de la pile en simple, double longueur et non signée. Nous avons également vu ." qui sert à afficher une chaîne de caractère terminée par " . Le mot CR effectue un retour chariot : passage en début de ligne suivante.

Voyons l'outil de base pour réaliser toute manipulation de l'écran ; c'est le mot EMIT qui affiche à l'écran le caractère dont le code ASCII est sur le sommet de la pile.

Nous attirons tout de suite l'attention sur la différence entre . et EMIT.

Exemple :

```
65 Return OK
. Return 65 OK
65 Return OK
EMIT Return A OK
```

Le mot CR est défini à partir de EMIT :

```
: CR 13 EMIT; Return OK
```

Ce type de mot de contrôle de l'écran est toujours défini de la même façon.

Exemple : Affichage d'un caractère blanc :

```
: SPACE 32 EMIT; Return OK
```


Pour afficher un nombre variable de blancs, il suffit de mettre ce nombre sur la pile et d'utiliser le mot SPACES défini comme suit :

```
: SPACES 0 DO SPACE LOOP ; Return OK
```

Pour vous détendre, essayez de créer, en utilisant les caractères de contrôle disponibles sur votre terminal, les mots suivants :

— PAGE qui vous affiche un écran blanc
— BEEP qui émet un bip sonore
— SETAB qui pose une tabulation horizontale
— TAB qui positionne le curseur à la prochaine tabulation horizontale posée.

Voyons un exercice un peu plus élaboré :
afficher à l'écran n caractères dont les codes ASCII sont contenus dans une zone mémoire commençant à une adresse donnée ADR.

```
: TYPE        ( Adr n --- )
  0 DO        ( Adr --- )
  DUP @       ( Adr  code ASCII --- )
  EMIT        ( Adr --- )
  1+
  LOOP        ( Adr+1 --- )
  DROP        ( --- )
; Return OK
```

Voici une autre application du clavier et de l'écran.

```
: JEU CR
  BEGIN
  32  WORD                 ( Transfert du mot à la fin du dictionnaire )
  HERE  1+  C@             ( Valeur du premier caractère )
        WHILE              ( Test de fin )
        HERE  COUNT        ( Préparation des paramètres pour TYPE )
        TYPE               ( Impression du mot )
        CR
  REPEAT
; [Return] OK
JEU A LA MAISON   [Return]
A
LA
MAISON
OK
```

L'interpréteur détecte et exécute successivement les mots résidant dans l'input stream. La détection est faite grâce au séparateur, ici le caractère blanc. Le mot que nous venons de définir inspecte au moment de son exécution le restant de l'input stream, fait le même travail de détection que l'interpréteur et affiche chaque mot qu'il a pu isoler sur une ligne de l'écran. La condition d'arrêt est la rencontre d'un caractère nul.

Un autre mot qui s'avère très utile est –TRAILING qui recalcule la longueur d'une chaîne en éliminant les caractères blancs en fin de chaîne.

```
-TRAILING        ( Adr u1 --- Adr u2 )
```

Formats

Nous allons examiner le formattage des nombres en double longueur non signés, nous verrons par la suite comment faire pour se ramener systématiquement à ce cas de figure.

La fonction du formattage est de construire la chaîne de caractères à imprimer qui représente la valeur située au sommet de la pile dans la base courante.

L'expression du format d'impression est délimitée par les mots <# et #>. Le format le plus simple consiste à convertir un nombre (double longueur) au sommet de la pile en son expression littérale sous forme de codes ASCII. C'est la fonction du mot #S.

Le mot UD. est défini de la façon suivante:

```
: UD. <# #S #> TYPE ; [Return] OK
```

Vous remarquerez que le format ayant construit la chaîne de caractères voulue, le mot #> prépare sur la pile les paramètres nécessaires au mot TYPE qui effectue l'impression.

Il est à noter que le mot #S ne produira pas de zéros non significatifs mais toujours au moins un chiffre.

Pour formatter plus finement un nombre il faut avoir la possibilité de convertir séparément chacun des chiffres; c'est la fonction du mot # qui place l'expression ASCII du chiffre le moins significatif dans la chaîne à éditer et le retranche du nombre.

Exemple:

```
: DISP2 <# # # #> TYPE ; [Return] OK
  325. DISP2 [Return] 25 OK        ( --- )
```

Cette possibilité de coder un à un des chiffres permet d'intercaler des caractères entre les chiffres grâce au mot HOLD, précédé du code ASCII que l'on veut intercaler.

Exemple:

Édition après calcul en virgule fixe à deux décimales.

```
: .# <# # # 46 HOLD #S #> TYPE   ; [Return] OK
123459. .# [Return] 1235.49 OK
```

Rappelons que l'interpréteur détecte les valeurs en double longueur par un point qui les suit (voir exemple précédent).

N'essayez pas de faire le rapprochement entre la notion FORTH de format et celle des langages évolués courants.

FORTH vous permet par exemple de changer de base à l'intérieur même du formattage: ceci est dû au fait que la notion de construction de chaîne de caractères n'est pas voilée.

Exemple: Affichage de l'heure ayant un nombre de secondes au sommet de la pile.

```
: SEXTAL 6 BASE ! ; Return OK
: :00 # SEXTAL # DECIMAL 58 HOLD ; Return OK
: SEC <# :00 :00 #S #> TYPE ; Return OK
14875. SEC Return 4:07:55 OK
```

Le mot SIGN va nous permettre de travailler sur des nombres signés. Il teste le signe de la troisième cellule de la pile et insère son code dans la chaîne de caractère en construction.

Les différents types d'utilisation du formattage sont:

Type de nombres à éditer	*Préparation*
32 bits non signé	---
31 bits signé	SWAP OVER DABS
16 bits non signé	0
16 bits signé	DUP ABS 0

Mot	*Syntaxe*	*Définition*
@	@(Ad --- Val)	Lit, sur 16 bits, la valeur contenue à l'adresse mémoire située au sommet de la pile, et la range au sommet de la pile
?	? (Ad ---)	Lit, sur 16 bits, la valeur contenue à l'adresse mémoire située au sommet de la pile, et l'affiche à l'écran.
HEX	HEX	La base courante devient hexadécimale.
DECIMAL	DECIMAL	La base courante devient décimale.
!	! (Val Ad ---)	Range, à l'adresse mémoire située au sommet de la pile, la valeur sur 16 bits qui la suit dans la pile. Ces deux éléments sont consommés dans l'opération
C@	C@ (Ad --- Val)	Lit, sur 8 bits, la valeur contenue à l'adresse mémoire située au sommet de la pile, et la range sur la pile.
C!	C! (Val Ad ---)	Range, à l'adresse mémoire située au sommet de la pile, la valeur sur 8 bits qui la suit dans la pile.
CONSTANT	Val CONSTANT Nom	Définit une constante appelée "Nom", et lui affecte la valeur "Val".
VARIABLE	Val VARIABLE Nom	Définit une variable appelée "Nom", et lui affecte la valeur "Val".
DUP	DUP (n --- n n)	Duplique le sommet de la pile.
DROP	DROP (n ---)	Dépile la valeur située au sommet de la pile.
SWAP	SWAP (n1 n2 --- n2 n1)	Intervertit les deux valeurs situées au sommet de la pile.
OVER	OVER (n1 n2 --- n1 n2 n1)	Recopie le deuxième élément de la pile au sommet de la pile.
ROT	ROT (n1 n2 n3 --- n2 n3 n1)	Effectue une permutation circulaire des trois cellules du sommet de la pile.
2DUP	2DUP (d --- d d)	Duplique un mot en double longueur sur la pile.
2DROP	2DROP (d ---)	Dépile un mot en double longueur.
2SWAP	2SWAP (d1 d2 --- d2 d1)	Intervertit les deux mots en double longueur au sommet de la pile.

Mot	Syntaxe	Définition
2OVER	2OVER (d1 d2 --- d1 d2 d1)	Recopie le deuxième élément, en double longueur, de la pile, au sommet de la pile.
2ROT	2ROT (d1 d2 d3 --- d2 d3 d1)	Effectue une permutation circulaire des trois mots en double longueur situés au sommet de la pile.
<#		Début de formattage des nombres. Attend un nombre signé sur 32 bits sur la pile.
#		Chaque # déclenche la production d'un chiffre. Des zéros seront systématiquement produits s'il ne reste pas assez de chiffres.
#S		Format libre : chaque chiffre est converti en un caractère.
HOLD	HOLD (c ---)	Insère un caractère dont le code ASCII est sur la pile dans le formattage.
SIGN		Insère un signe moins, si le troisième nombre dans la pile est négatif. Généralement utilisé juste avant #>.
#>		Fin du formattage. L'adresse de la chaîne produite, et le nombre de caractères sont dans la pile, prêts pour TYPE.
>R	>R ds:(n ---) rs:(--- n)	Transfert la valeur au sommet de la pile de données vers la pile de return.
R>	R> ds:(--- n) rs:(n ---)	Transfert la valeur au sommet de la pile de return vers la pile de données.
+	+ (n1 n2 --- n1+n2)	Addition des deux éléments au sommet de la pile. Les opérandes sont consommés et le résultat est rangé au sommet de la pile.
–	– (n1 n1 --- n1–n2)	Soustraction des deux éléments au sommet de la pile.
*	* (n1 n2 --- n1*n2)	Multiplication des deux éléments au sommet de la pile. Le résultat est rangé au sommet de la pile.
/	/ (n1 n2 --- E(n1/n2))	Division entière des deux éléments au sommet de la pile. Le résultat est rangé sur la pile.
MOD	MOD (n1 n2 --- R(n1/n2))	Reste de la division entière des deux éléments situés au sommet de la pile.

Mot	Syntaxe	Définition
/MOD	/MOD (n1 n2 --- E(n1/n2) R(n1/n2))	Effectue une division des deux éléments situés au sommet de la pile. Retourne le quotient et le reste.
*/	*/ (n1 n2 n3 --- n1*n2/n3)	Multiplication du deuxième par le troisième élément de la pile, puis division du résultat par le premier élément. Calcul intermédiaire sur 32 bits.
ABS	ABS (n --- |n|)	Retourne la valeur absolue du sommet de la pile.
NEGATE	NEGATE (n--- –n)	Change le signe du sommet de la pile.
MIN	MIN (n1 n2---min)	Retourne le minimum de deux paramètres chargés dans la pile.
MAX	MAX (n1 n2---max)	Retourne le maximum de deux paramètres chargés dans la pile.
1+	1+ (n --- n+1)	Ajoute 1 à la valeur située au sommet de la pile.
1–	1– (n --- n–1)	Retranche 1 à la valeur située au sommet de la pile.
2+	2+ (n --- n+2)	Ajoute 2 à la valeur située au sommet de la pile.
2–	2– (n --- n–2)	Retranche 2 à la valeur située au sommet de la pile.
2*	2* (n --- 2*n)	Multiplication par 2 du sommet de la pile (décalage arithmétique de 1 position à gauche).
2/	2/ (n --- n/2)	Division entière par deux du sommet de la pile (décalage arithmétique de 1 position vers la droite).
=	= (n1 n2 --- b)	Teste l'égalité de deux paramètres en simple longueur (16 bits). Retourne un booléen sur la pile.
<	> (n1 n2 --- b)	Comparaison de deux paramètres en simple longueur (16 bits). Retourne un booléen sur la pile.
>	> (n1 n2 --- b)	Comparaison de deux paramètres en simple longueur (16 bits). Retourne un booléen sur la pile.
0=	0= (n --- b)	Teste la nullité d'un paramètre en simple longueur (16 bits). Retourne un booléen sur la pile.
0>	0> (n --- b)	Compare un paramètre à zéro. Retourne un booléen sur la pile.

Mot	Syntaxe	Définition
0<	0< (n --- b)	Compare un paramètre à zéro. Retourne un booléen sur la pile.
AND	AND (n1 n2 --- n1 AND n2)	AND logique de deux paramètres en simple longueur.
OR	OR (n1 n2 --- n1 OR n2)	OR logique de deux paramètres en simple longueur.
NOT	NOT (b --- b)	Inverse un booléen. Équivalent à 0=.
XOR	XOR (n1 n2 --- n1 XOR n2)	Ou exclusif de deux paramètres en simple longueur.
D+	D+ (d1 d2 --- d1+d2)	Addition sur 32 bits.
D−	D− (d1 d2 --- d1−d2)	Soustraction sur 32 bits.
DNEGATE	DNEGATE (d --- −d)	Changement de signe sur 32 bits.
DABS	DABS (d --- |d|)	Valeur absolue sur 32 bits.
DMAX	DMAX (d1 d2 --- Max)	Maximum de deux valeurs sur 32 bits.
DMIN	DMIN (d1 d2 --- Min)	Minimum de deux valeurs sur 32 bits.
D=	D= (d1 d2 --- b)	Comparaison sur 32 bits. Retourne un flag sur la pile.
D0=	D0= (d1 --- b)	Teste la nullité d'un paramètre sur 32 bits.
D<	D< (d1 d2 --- b)	Comparaison sur 32 bits.
D.	D. (d ---)	Affichage d'un nombre sur 32 bits.
DU<	DU< (ud1 ud2 --- b)	Test sur des nombres non-signés sur 32 bits.
M+	M+ (d n --- d)	Additionne un nombre sur 32 bits à un nombre sur 16 bits. Retourne un résultat sur 32 bits.
M/	M/ (d n --- n)	Divise un nombre sur 32 bits par un nombre sur 16 bits. Le résultat est sur 16 bits. Toutes les valeurs sont signées.
M*	M* (n1 n2 --- d)	Multiplie deux nombres sur 16 bits et fournit un résultat sur 32 bits.

62

Mot	Syntaxe	Définition
M*/	M*/ (d n u --- d)	Multiplie un nombre sur 32 bits par un nombre sur 16 bits, puis divise le résultat, sur 48 bits, par un nombre sur 16 bits. Le résultat est sur 32 bits.
S−>D	S−>D (n --- d)	Convertit un nombre de 16 bits à 32 bits.
U.	U. (u ---)	Affiche un nombre non signé en simple longueur.
U*	U* (u1 u2 --- ud)	Multiplie deux nombres non-signés en simple longueur, et retourne un nombre signé en double longueur.
U/MOD	U/MOD (ud u1 --- u2 u3)	Divise un nombre non signé sur 32 bits par un nombre non-signé sur 16 bits. Retourne le quotient et le reste, en simple longueur, non-signés.
U<	U< (u1 u2 --- b)	Compare deux nombres non-signés en simple longueur.
IF xxx ELSE yyy THEN zzz	IF (b ---)	Si le booléen au sommet de la pile est vrai au moment de l'exécution de IF, la clause xxx est exécutée. Sinon, c'est yyy qui est exécutée. Dans tous les cas, l'exécution continuera avec zzz.
DO...LOOP	DO (n1 n2 ---) LOOP (---)	Structure de boucle finie. La clause encadrée par DO et LOOP sera effectuée tant que index est strictement inférieur à fin, avec incrémentation automatique de index à l'exécution de LOOP.
DO...+LOOP	DO (n1 n2 ---) +LOOP (n ---)	Même chose, mais l'index est augment de la valeur présente sur la pile au moment de l'exécution de +LOOP.
BEGIN... AGAIN	BEGIN (---) AGAIN (---)	Boucle infinie.
BEGIN... UNTIL	BEGIN (---) UNTIL (b ---)	Boucle infinie, avec possibilité de sortie si le booléen présent sur la pile au moment de l'exécution de UNTIL est vrai.
BEGIN xxx WHILE yyy REPEAT	BEGIN (---) WHILE (b ---)	Boucle infinie. La clause xxx est toujours exécutée. Si le booléen présent sur la pile au moment de l'exécution de WHILE est VRAI, la clause yyy sera elle aussi exécutée. Le bouclage s'arrête si ce booléen est FAUX.
KEY	KEY (--- c)	Retourne sur la pile la valeur ASCII du prochain caractère accessible issu du périphérique courant d'entrée.

63

Mot	Syntaxe	Définition
EXPECT	EXPECT (Adr u ---)	Attend que u caractères soient disponibles ou qu'un retour chariot soit rencontré, et les range à l'adresse indiquée.
WORD	WORD (c --- Adr)	Lit un mot dans l'input stream, en utilisant le caractère indiqué comme délimiteur. La chaîne est alors rangée à l'adresse pointée par HERE, avec sa longueur en tête. L'adresse de rangement est retournée sur la pile.
TEXT	TEXT (c ---)	Lit un mot dans l'input stream, en utilisant le caractère indiqué comme délimiteur, remet le PAD à blanc, et y range le mot trouvé.
DUMP	DUMP (Adr n ---)	Affiche le contenu des n premières cellules mémoire à partir de l'adresse Adr.
COUNT	COUNT (Adr --- Adr+1 u)	Prépare les paramètres nécessaires pour TYPE. A l'adresse fournie, on trouvera la longueur de la chaîne.
EMIT	EMIT (c ---)	Affiche le caractère dont le code ASCII se trouve au sommet de la pile.
-TRAILING	-TRAILING (Adr u1 --- Adr u2)	Recalcule la longueur d'une chaîne éliminant tous les caractères blancs en fin de chaîne.
TYPE	TYPE (Adr u ---)	Affiche u caractères à l'écran, à partir d'une adresse donnée. Ne pas confondre avec DUMP.

EXERCICE 1:

Quel est l'état de la pile après les séquences suivantes:

a) 1 2 3 ROT SWAP OVER ROT DROP

b) 1 2 3 OVER + ROT OVER * SWAP DROP

c) 1 2 3 * OVER ROT DROP /

d) 1 2 3 + SWAP 5 - *

e) 21 43 56 + 120 /MOD SWAP . . . .

f) 1 2 3 4 2OVER + SWAP - 2DUP

EXERCICE 2:

Les variables étant empilées comme indiqué, écrivez un mot FORTH qui permette d'évaluer chacune des expressions suivantes:

a) (a b c d ---)

((a+c)*b)/(c+d)

b) (a b c d e ---)

((c+e)*(a+c))/((b+d)*(a+c)/e)

c) (a b c ---)

(((a+c)*b)+(c+b))/((a+c)*a)

EXERCICE 3:

a) Écrire un mot FORTH qui calcule le minimum de N valeurs situées en mémoire à partir de l'adresse Ad.

Début : (Ad n ---)

FIN : (--- Min)

b) Écrire un mot qui lui calcule le maximum de ces valeurs.

EXERCICE 4:

a) Écrivez la soustraction de deux nombres situés sur la pile sans utiliser -.

b) Écrivez */MOD en FORTH.

EXERCICE 5

a) Écrivez un mot qui remplit de 0 une zone mémoire de N cellules commençant à l'adresse Ad.

(Ad N ———)

b) Écrivez un mot qui incrémente de 1 cette même zone mémoire. (Ad N ———).

EXERCICE 6:

a) Écrivez un mot qui calcule le PGCD (Plus Grand Commun Diviseur) des deux nombres entiers simple longueur du sommet de la pile.

(N1 N2 ——— PGCD)

b) De même écrivez un mot qui calcule le PPCM (Plus Petit Commun Multiplieur).

(N1 N2 ——— PPCM)

EXERCICE 7:

Définissez le mot DIV qui effectuera la division réelle de deux nombres entiers simple longueur, avec quatre chiffres après la virgule.

La syntaxe à l'utilisation devra être de la forme:

10 3 DIV [Return] 3.3333 OK

On utilisera la structure DO ——— LOOP, en multipliant à chaque pas le reste par 10, et en effectuant les tests adéquats pour afficher le point décimal et pour sortir de la boucle (LEAVE).

EXERCICE 8:

Reprendre l'exercice précédent en utilisant la structure BEGIN ——— UNTIL.

EXERCICE 9:

Reprenez l'exercice 7 avec la structure BEGIN ——— WHILE ——— REPEAT.

EXERCICE 10:

Définissez le mot CONV qui affiche la valeur binaire du nombre simple longueur situé sur la pile (sans utiliser BASE, tricheur !).

Exemple: 22 CONV [Return] 10110 OK

EXERCICE 11:

Définissez le mot PREMIER qui affiche en décimal tous les nombres premiers entre 0 et 100.

EXERCICE 12:

Définissez le mot ANALYSE qui affichera le nombre de voyelles, le nombre de consommes et le nombre de blancs contenus dans la chaîne de caractères qui le suit.

La syntaxe d'utilisation sera donc:

ANALYSE "chaîne de caractères" [Return]

IV

LES VOCABULAIRES DE BASE

IV.1. L'ÉDITEUR

IV.1.1. Nécessité de garder les sources

Vous avez certainement constaté que, jusqu'à présent, il vous est impossible de modifier la définition d'un mot sans avoir à la redéfinir complètement.

FORTH est un compromis entre la compilation et l'interprétation.

On pourrait donc imaginer, dans la lignée des langages interprétés (BASIC par exemple), la possibilité de pouvoir "appeler" le texte d'une définition de mot, et d'y apporter des modifications dont la prise en compte ne remette pas en cause le reste de l'environnement.

Cette exigence est incompatible avec un langage évolutif comme FORTH. Le principal obstacle est lié au mode d'implantation en mémoire du dictionnaire : les définitions, de longueur variable, s'accumulent consécutivement. Tout changement dans la longueur d'une définition peut entraîner de grosses manipulations de mémoire et d'inextricables modifications de pointeurs.

C'est pour cette raison que l'on ne peut disposer des outils de rédaction de mots dérivés des interpréteurs.

La solution est de conserver dans une zone mémoire réservée à cet effet le texte source des définitions. Des procédures permettront de traiter ces textes et de lancer leur semi-compilation, afin de mettre à jour le dictionnaire.

Il faut bien entendu trouver un compromis [illegible] e le [illegible] s d'[illegible] et la mémoire interne disponible pour répartir le texte entre la mémoire vive utilisateur et la mémoire auxiliaire de masse (disques principalement).

Pour avoir la plus grande souplesse, il est convenu de découper le texte en pages et de confier à la machine la gestion de ces pages de sorte que l'accès soit totalement transparent pour l'utilisateur.

Les spécialistes reconnaîtront là une gestion simplifiée de mémoire virtuelle.

Les pages de texte se présentent donc comme un livre que peut feuilleter l'utilisateur sans avoir à se soucier de l'emplacement physique d'une page au moment de l'appel.

IV.1.2. Principe de fonctionnement

Dans la plupart des FORTH sur micro-ordinateurs, le nombre des pages résidentes en mémoire utilisateur est fixé et la taille d'une page correspondra à la taille de l'écran physique du terminal.

Pour des raisons de transparence, il est nécessaire d'implanter un algorithme de recouvrement des pages.

La numérotation des pages est choisie en relation avec les numéros d'ordre d'implantation physique sur le disque.

Notons tout de suite qu'une machine puissante sera capable, de par son système d'exploitation, d'offrir des possibilités plus étendues (appel par nom par exemple).

Pour présenter clairement l'algorithme d'implantation en mémoire d'une nouvelle page, nous supposerons reconnus les mots FORTH suivants, dont les définitions dépendent trop du système d'exploitation de la machine pour être détaillées ici :

NMAX	Constante donnant le nombre maximum de pages pouvant résider simultanément en mémoire centrale.
NCHARGE	Variable donnant le nombre de pages résidentes en mémoire centrale, à tout instant.
PREMS	Variable donnant l'emplacement destiné à la prochaine page.

FLAGPREMS	Mot retournant sur la pile un flag indiquant si la page se trouvant à l'adresse PREMS a été modifiée sans être sauvegardée.
MODIFPREMS	Mot mettant à jour la variable PREMS après un chargement.
MEM<—DISC	Mot exécutant un chargement de page en mémoire centrale à l'adresse spécifiée au sommet de la pile.
DISC<—MEM	Mot exécutant la copie sur disque de la page spécifiée au sommet de la pile.

L'algorithme s'écrit alors de façon très simple :

```
: CHARGE
        NCHARGE @  NMAX @  <
        IF
              NCHARGE DUP @   1+ SWAP  !
        THEN
              FLAGPREMS
              IF PREMS @ DISC<—MEM THEN
        ELSE
        PREMS @  MEM<—DISC
        MODIFPREMS
; [Return] OK
```

C'est bien sûr dans MODIFPREMS que réside le cœur de l'algorithme de recouvrement. Les grandes orientations dans la résolution de ce problème sont :

— *Allocation FIFO* (First In – First Out) : C'est la plus ancienne des pages résidentes que l'on remplace.

— *Allocation statistique* : c'est la moins fréquemment utilisée que l'on remplace.

— *Allocation hiérarchisée* : c'est la page de priorité la plus faible que l'on remplace.

Remarquons toutefois pour finir que les systèmes d'exploitation utilisent en général une combinaison de ces différentes méthodes, l'algorithme

ɔou᷈ mêr ᷈volu ᷈ynaı ᷈ :mei in (, misc.. ..s te...,.. de réponses.

En FORTH, on dispose des commandes de gestion de page suivantes :

LIST	Cherche une page en mémoire centrale, la charge si elle n'est pas résidente, et affiche son texte à l'écran.
LOAD	Cherche une page en mémoire centrale, la charge si elle n'est pas résidente, et déroute son texte vers l'interpréteur.
FLUSH	Copie sur disque toutes les pages résidentes en mémoire centrale qui ont été modifiées.
EMPTY–BUFFERS	Initialise la mémoire de pages.

IV.1.3. Les commandes

Toutes les commandes décrites ci-après sont regroupées dans le vocabulaire EDITOR. Avant de vous plonger dans la lecture de ce chapitre, assurez vous dès maintenant que ce vocabulaire est bien disponible sur votre machine.

Sous une forme évoluée, les éditeurs peuvent s'apparenter à de véritables traitements de texte. Nous n'aborderons ici qu'un type d'éditeurs assez rustiques, souvent utilisés dans le monde de l'informatique.

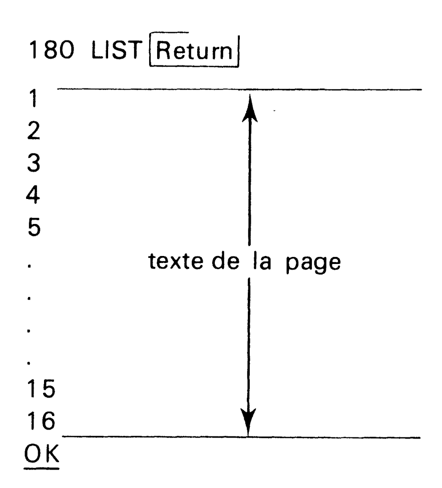
Les commandes de notre éditeur vont utiliser deux buffers spécialisés : l'INSERT BUFFER (buffer d'insertion), et le FIND BUFFER (buffer de recherche).

De façon classique, les déplacements à l'intérieur d'une ligne se font par l'intermédiaire d'un pointeur de position représenté à l'écran par un caractère spécial inséré avant le caractère pointé.

Avant tout travail sur une page, il faut appeler la page choisie à l'écran, à l'aide de la commande LIST.

La page va alors apparaître à l'écran, toutes les lignes étant numérotées dans la marge.

Vous pouvez éventuellement mettre tout le contenu initial de la page à blanc en utilisant la commande WIPE. Cette commande n'agit que sur la page sélectionnée grâce à LIST.



Pour modifier le contenu d'une ligne, vous devez d'avord la sélectionner à l'aide du mot T (Take).

Exemple :

5 T [Return]
↑ 5 OK

Ce mot affiche le contenu de la ligne choisie, et positionne le pointeur en début de ligne, avant le premier caractère.

Pour écrire dans la ligne, on utilise le mot P (Put) :

P TOTO [Return] OK

5 T [Return]
↑ TOTO 5 OK

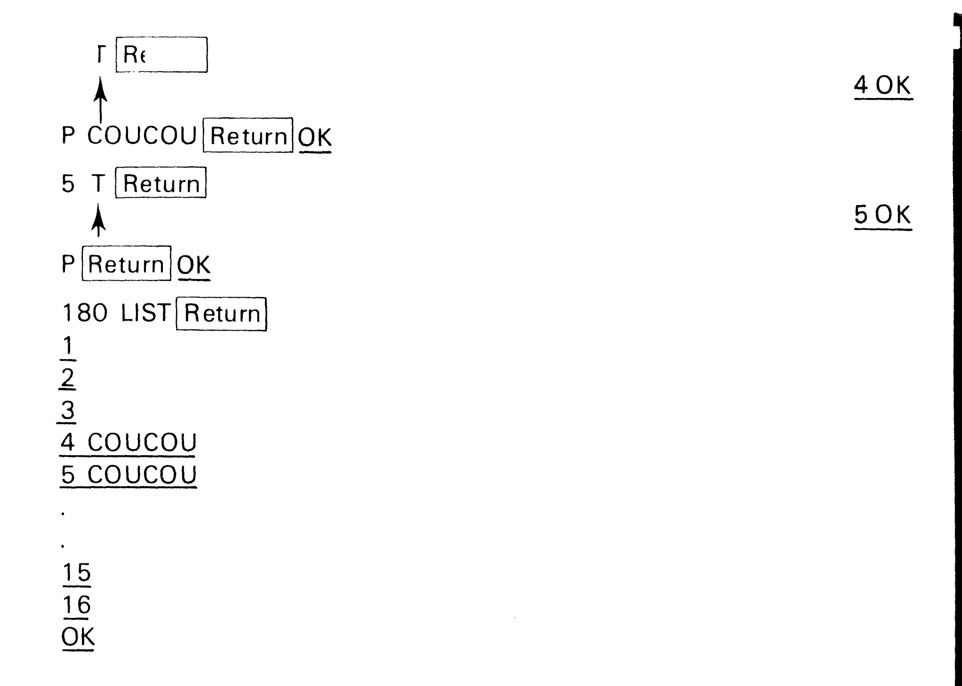
Le mot P place la chaîne de caractères qui le suit dans l'INPUT BUFFER, puis copie ce dernier au début de la ligne courante (le contenu précédent de la ligne est perdu).

Une méthode pour mettre à blanc une ligne est de taper la commande P suivie d'un espace blanc.

P [Return] OK

Une façon d'utiliser le mot P consiste à le faire suivre immédiatement de [Return]. L'INPUT BUFFER n'est pas modifié, et son contenu est placé au début de la ligne courante.

Cette astuce peut être très utile pour copier le contenu de l'INPUT BUFFER d'une ligne sur une autre.



T [Re...] 4 OK

P COUCOU [Return] OK

5 T [Return] 5 OK

P [Return] OK

180 LIST [Return]

1
2
3
4 COUCOU
5 COUCOU
.
.
15
16
OK

Le déplacement dans la ligne se fait grâce au mot F (Find), qui positionne le pointeur après la première occurrence d'une chaîne donnée, en commençant la recherche à la position courante du pointeur.



4 T [Return]

↑COUCOU 4 OK

F CO [Return]

CO↑COU 4 OK

La commande I permet d'insérer une chaîne dans une ligne après la position courante du pointeur. Il n'y a pas de report d'une ligne sur une autre si l'insertion provoque un débordement dans une ligne.



I UCO [Return]

COUCO↑COU 4 OK

I [Return]

COUCOUCO↑COU 4 OK

Encore une fois, vous aurez noté que les commandes suivies immédiatement de [Return] travaillent directement sur le contenu des buffers sans les modifier.

Après avoir localisé une chaîne dans la ligne courante grâce à F, il est possible de la supprimer par la commande E:

E [Return]

COUCO↑UCOU 4 OK

Ces deux actions sont combinées dans la commande D:

D UC [Return]

COUCO↑OU 4 OK

Une commande plus élaborée permet d'effacer une partie de la ligne: TILL efface tout le contenu de la ligne depuis la position courante du curseur, jusqu'à et y compris la première occurence d'une chaîne donnée.

4 T [Return]

↑COUCOU 4 OK

P MAIS OU ET DONC OR NI CAR [Return] OK

F ET [Return]

MAIS OU ET↑DONC OR NI CAR 4 OK

TILL OR [Return]

MAIS OU ET ↑ NI CAR 4 OK

La commande XCH permet de faire des remplacements de chaînes dans une ligne. Elle place une chaîne donnée dans l'INSERT BUFFER, et la replace dans la ligne courante, en remplacement de la chaîne couramment pointée, contenue dans le FIND BUFFER. L'opération se passe en deux temps: La chaîne contenue dans le FIND BUFFER est supprimée de la ligne, puis celle contenue dans l'INSERT BUFFER est insérée à la position courante du pointeur.

Comme dans les cas précédents, le paramètre est optionnel. XCH peut travailler directement sur le contenu courant de l'INSERT BUFFER.

Exemple:

5 T [Return] 5 OK

P LE CHASSEUR CHASSE [Return] OK

F CHASS [Return]

LE CHASS↑EUR CHASSE 5 OK

XCH PECH [Return]

LE PECH↑EUR CHASSE 5 OK

En ' né, un au r...pitulа... ..es d...entes ...ctions.

Syntaxe	*Action sur les buffers*	*Description*
P (ut)	chaîne —> I.B.	Introduction d'une ligne.
F (ind)	chaîne —> F.B.	Recherche du contenu de F.B. dans une ligne.
I (nsert)	chaîne —> I.B.	Insertion du contenu de I.B. dans une ligne.
E (rase)		Suppression du contenu de F.B. dans une ligne.
D (elete)	chaîne —> F.B.	Combinaison de F et E.
XCH	chaîne —> I.B.	Combinaison de E et I.

Lorsque l'on introduit un texte dans une page vierge, il est agréable de voir les lignes s'incrémenter automatiquement.

Le mot P ne répond pas à ce souhait, puisqu'il suppose que la ligne a été au préalable sélectionnée à l'aide de T. Par ailleurs il ne touche pas au pointeur de la ligne courante.

Le mot U comble ces lacunes : il insère la chaîne dans la page après la ligne couramment pointée en décalant toutes les lignes suivantes vers le bas, et incrémente le pointeur de ligne.

Les lignes qui sortent de l'écran sont perdues.

La commande X a la fonction inverse : Elle supprime la ligne courante, et décale les suivantes vers le haut d'un cran.

La ligne courante n'est toutefois pas perdue : Elle se trouve dans l'INSERT BUFFER, prête à être replacée.

WIPE, comme nous l'avons vu, a une action plus radicale puisqu'il efface tout l'écran.

Le mot L, abrégé de LIST, édite la page courante, sans avoir besoin de préciser son numéro.

Pour passer à une page voisine, on utilisera N et B qui incrémente et décrémente respectivement le numéro de page courante.

Pour déplacer une ligne d'un écran à un autre, on peut utiliser le mot M (MOVE), qui opère sur la ligne courante, et nécessite deux paramètres : Le numéro de page destination, et le numéro de ligne dans cette page après laquelle il fera l'insertion.

S pour sa part sert à faire des recherches systématiques d'une chaîne sur des écrans consécutifs.

L... paramè... à lui fournir sont le numéro du dernier écran où s'opère la recherche, et la chaîne à chercher.

IV.2. L'ASSEMBLEUR

Vous avez remarqué que FORTH, langage évolué s'il en est, dispose d'un certain nombre de mots qui permettent de descendre à un niveau très proche de l'architecture de la machine. (ex : manipulation des cellules mémoire, accès aux ports d'E/S etc...).

Il existe cependant certains cas de figures où seul un véritable assembleur peut résoudre les problèmes par exemple :

— Vous souhaitez accélérer l'exécution de vos procédures en optimisant le code généré par élimination des indirections. C'est par exemple le cas de la connexion de périphériques à très fort débit, ou le traitement des interruptions.

— Vous avez besoin d'accéder aux registres internes du processeur : Le cas classique est le déroutement par modification du PC.

La nécessité de l'acces à l'assembleur étant entendue, nous allons satisfaire devant vous à ce besoin, dans le cadre de la philosophie FORTH.

Les BASIC standards apportent une réponse à ce problème de la façon suivante : La procédure écrite en langage machine étant résidante en mémoire et de préférence terminée par RET, une instruction BASIC permet de se dérouter sur cette routine en modifiant le contenu de PC, sous une forme assimilable à un CALL.

Il est souhaitable que cette routine ait été sérieusement testée au préalable car vous ne disposez en BASIC d'aucun outil évolué de type Éditeur-Assembleur pour la mettre au point.

La solution de base équivalente qu'offre FORTH est beaucoup plus souple car elle va permettre de générer le code machine depuis FORTH.

Quels sont les outils dont nous avons besoin pour cela ?

Un premier devra permettre de signaler à l'interpréteur que ce qui suit dans la définition du mot n'est plus du FORTH à proprement parler, mais du code directement exécutable.

Un second, et dernier, devra permettre d'intégrer le dit code dans le dictionnaire, à la suite dans la définition.

Arrêtons-nous quelques instants sur le comportement assez particulier de ce mot. Nous allons voir qu'il a en fait deux actions différentes selon que nous sommes dans une phase de compilation, ou dans la phase d'exécution du mot qui le contient.

Pendant la phase de compilation du mot considéré, ;CODE va simplement signifier à l'interpréteur que la définition du mot est terminée, sans toutefois la valider comme l'aurait fait ; . Il est donc alors possible de compléter cette forme provisoire de la définition par des instructions machine.

C'est là qu'intervient notre deuxième outil, à savoir les mots C, et , qui transfèrent le contenu du sommet de la pile, respectivement sur un et deux octets, à la suite du dictionnaire.

A l'exécution du mot ainsi défini, la rencontre de ;CODE va signaler à l'interpréteur que la suite de la définition n'est composée que d'instructions machine, à exécuter immédiatement.

Il est très important d'ores et déjà de bien assimiler ce double comportement. Nous le retrouverons plus détaillé dans les chapitres suivants.

La structure générale d'une définition contenant du code-machine est donc :

```
: MOT (liste de mots);CODE xx C, xxxx , (etc...)
```

où xx et xxxx représentent les codes objets exprimés dans la base courante.

Vous constaterez qu'il n'y a pas de mot de fin de définition après votre code, et que vous devez donc prendre en charge vous même le retour à l'interpréteur FORTH.

Pour ce faire, il vous suffit de terminer votre code par celui d'un saut au point d'entrée de l'interpréteur FORTH.

D'autre part il est bien évident que vous devrez dans bien des cas commencer par sauvegarder tous les registres avant de les utiliser, et les restaurer à leur valeur initiale avant de retourner à l'interpréteur.

Exemple :

Prenons le cas de l'assembleur Z 80 bien connu, et donnons nous comme problème très simple la mise à zéro de la zone mémoire qui s'étend de FA10 à FAB0. (Nous vous conseillons de vous assurer au préalable que cette zone est inoccupée...).

Le programme Assembleur correspondant s'écrit :

```
LD HL, FA10    21 10 FA
LD DE, FA11    11 11 FA
LD BC,A0       01 A0 00
LD (HL),0      36 00
LDIR           ED B0
```

Supposons que le point d'entrée de l'interpréteur ait comme adresse $6000. Pour que votre procédure soit intégrable à FORTH, elle devra se présenter sous la forme :

```
PUSH BC     C5
PUSH DE     D5
PUSH HL     E5
.
.
corps de la procédure ci-dessus
.
.
POP HL      E1
POP DE      D1
POP BC      C1
JP 6000     63 00 60
```

Vous définirez donc finalement le mot de la manière suivante :

```
HEX [Return] OK
: ZERO ;CODE
            C5 C, D5 C, E5 C, 21 C, FA10 , 11 C, FA11,
            01 C, 00A0 , 36 C, 00 C, EDB0 , E1 C, D1 C,
            C1 C, C3 C, 6000 ,
[Return] OK
```

Vous pouvez maintenant être tenté d'exécuter ZERO.

Malheureusement, un message d'erreur apparaît qui indique que la machine n'a pas trouvé une définition correcte dans le dictionnaire. Il faut savoir en effet qu'il existe un flag pour chaque mot dans le dictionnaire qui indique la validité de sa définition.

L'interpréteur construit une définition de mot dans le dictionnaire en analysant séquentiellement tous les mots de la ligne. Au cours de cette analyse, il se peut qu'un mot soit rejeté et donc le début de la définition, bien que présent dans le dictionnaire est inutilisable.

C'est la raison pour laquelle le flag de validité n'est positionné que lors de la rencontre du mot ; .

Vous avez constaté qu'il n'y avait pas de mot de fin de définition pour un mot englobant du code. Vous comprenez donc pourquoi le flag n'est pas positionné.

Qu'à cela ne tienne, un mot FORTH permet de forcer artificiellement le flag de validité du dernier mot du dictionnaire : c'est le mot SMUDGE.

```
SMUDGE [Return] OK
ZERO [Return] OK
```

Le mot ZERO a été cette fois bien exécuté.

Cette façon d'intégrer du code dans une définition bien qu'opérationnelle, est pénible à mettre en œuvre.

La première idée qui vient à l'esprit pour éliminer la phase de transcodage des codes objets correspondant aux mnémoniques est de créer les mots FORTH adéquat.

Par exemple :

```
LD HL, FA10        ( = 21 10 FA )
```

peut être facilement réalisé en créant le mot FORTH.

```
. LDHL 21 C, ; [Return] OK
```

Pour intégrer le code dans la définition, vous taperez cette fois :

```
......... LDHL FA10 , .........
```

Une solution consiste donc à disposer d'un vocabulaire ASSEMBLEUR complet, regroupant tous les mots de ce type.

Par exemple :

```
: PUSHBC    C5 C,     ; [Return] OK
: PUSHDE    D5 C,     ; [Return] OK
: PUSHHL    E5 C,     ; [Return] OK

: LDHL      21 C,     ; [Return] OK
: LDBC      11 C,     ; [Return] OK
: LD(HL)    36 C,     ; [Return] OK

: LDIR      EDB0 ,    ; [Return] OK

: JP        C3 C,     ; [Return] OK

: POPBC     E1 C,     ; [Return] OK
: POPDE     D1 C,     ; [Return] OK
: POPHL     C1 C,     ; [Return] OK
```


Notre exemple devient alors :

```
: ZERO ;CODE
                PUSHBC
                PUSHDE
                PUSHHL
                LDHL FA10 ,
                LDDE FA11 ,
                LDBC A0 ,
                LD(HL) 0 C,
                LDIR
                POPHL
                POPDE
                POPBC
                JP 6000 ,
SMUDGE [Return] OK
```

Cette solution est très répandue sur les micro-ordinateurs.

Vous imaginez facilement que, si vous ne disposez pas du vocabulaire ASSEMBLEUR, il vous sera très facile de le construire de cette façon.

Par contre, elle présente un certain nombre de faiblesses :

— il n'y a pas de contrôle syntaxique, sur le nombre d'opérandes par exemple

— il n'y a pas de possibilité d'utiliser des labels et il faut donc calculer à chaque fois avec précision les adresses absolues de branchement

— la syntaxe peut s'éloigner beaucoup de celle de l'assembleur. Vous devrez en effet définir non pas autant de mots qu'il y a de mnémoniques, mais autant de mots qu'il y a de codes reconnus.

Sachez également qu'il y a souvent un rapport étroit entre le nombre de codes reconnus et le produit du nombre de mnémoniques par le nombre total de registres internes.

Le niveau suivant dans la souplesse d'utilisation consiste à écrire un véritable assembleur en FORTH.

Les mnémoniques et leur syntaxe seront conservés et vous disposerez de tous les outils classiques (Labels, variables, constantes, messages d'erreur...).

Ce type de logiciel fait appel à de nombreux résultats théoriques, qui s'écartent largement du cadre de cet ouvrage.

De bonnes bases théoriques et une expérience approfondie de FORTH et de votre système doivent vous permettre cependant de résoudre le problème.

RESUMÉ DU CHAPITRE IV

Mot	*Syntaxe*	*Définition*
LIST	"N°page" LIST	Affiche à l'écran la page demandée, après l'avoir éventuellement chargée du disque, si elle n'était pas résidante en mémoire.
LOAD	"N°page" LOAD	Compile le contenu de la page demandée, après l'avoir éventuellement chargée du disque, si elle n'était pas résidente en mémoire.
FLUSH	FLUSH	Vide les block buffers après avoir sauvegardé sur disque ceux qui ont été modifiés.
EMPTY –BUFFERS	EMPTY–BUFFERS	Vide les block buffers.
EDITOR	EDITOR	Rend EDITOR le vocabulaire de contexte.
T	"N°ligne" T	Sélectionne la ligne demandée dans la page de travail, l'affiche à l'écran et positionne le curseur en début de ligne.
P	– P "chaîne"	Place "chaîne" dans l'insert buffer et remplace la ligne courante par ce dernier.
	– P [Return]	Vide l'insert buffer et la ligne courante.
	– P [Return]	Remplace la ligne courante par le contenu de l'insert buffer.
I	– I "chaîne"	Insert la chaîne "chaîne" après le curseur dans la ligne courante et dans l'insert buffer.
	– I [Return]	Insert le contenu de l'insert buffer avant le curseur.
E	E	Efface dans la ligne courante la première occurence de la chaîne de caractères contenue dans le find buffer.
D	D "Chaîne"	Efface dans la ligne courante la première occurence de la chaîne de caractère, après avoir placé cette dernière dans le Find Buffer.
F	– F "Chaîne"	Place la chaîne de caractères dans le Find Buffer, en recherche la première occurrence dans la ligne courante, et positionne le pointeur de ligne.
	– F [Return]	Recherche la première occurence de la chaîne de caractères contenue dans le Find Buffer dans la ligne courante, et positionne le pointeur de ligne.

Mot	*Syntaxe*	*Définition*
TILL	TILL "Chaîne"	Efface de la ligne courante tous les caractères compris entre la position du pointeur de ligne, et la première occurence de la chaîne de caractère.
XCH	XCH "Chaîne"	Échange dans la ligne courante la chaîne de caractère contenue dans le Find Buffer par celle qui suit la commande.
U	U "Chaîne"	Insère une ligne sous la ligne courante, en respectant les fonctionalité de P.
X	X	Supprime la ligne courante, et décale les suivantes vers le haut.
L	L	Liste à l'écran la page courante.
N	N	Incrémente de 1 le numéro de la page courante.
B	B	Décrémente de 1 le numéro de la page courante.
M	Page Ligne M	Insère la ligne courante de la page courante sous la ligne spécifiée dans la page spécifiée.
ASSEMBLEUR	ASSEMBLEUR	Rend le vocabulaire ASSEMBLEUR vocabulaire de contexte.
;CODE	: <name> .... ;CODE <mnémoniques>	Arrête la compilation, et permet de rentrer, soit des mnémoniques assembleur si l'on dispose du vocabulaire adéquat, soit directement des codes exécutables à l'aide des mots , et C, .
,	, (n ——)	Prend le contenu du sommet de la pile, le range dans la première cellule libre du dictionnaire, et incrémente de 2 le pointeur de fin du dictionnaire.
C,	C, (n ——)	Prend les 8 bits les moins significatifs de la valeur accessible au sommet de la pile, les range dans le premier octet libre du dictionnaire, et incrémente de 1 le pointeur de fin du dictionnaire.
SMUDGE	SMUDGE	Inverse le bit de validité de la dernière tête de chaîne créée dans le dictionnaire.

V

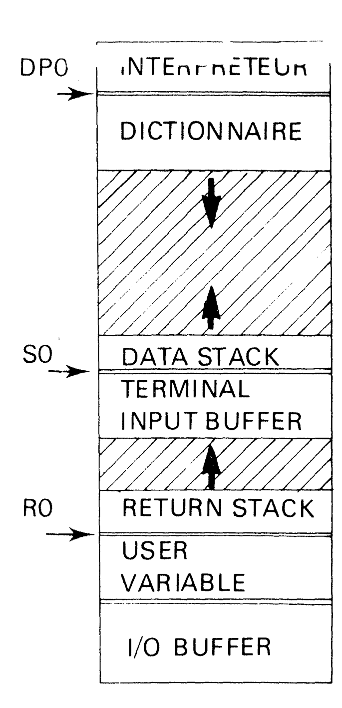
LA MÉCANIQUE DU LANGAGE

V.1. Anatomie

V.1.1. Memory-Map

Dans cette partie nous nous proposons d'aborder FORTH sous un éclairage purement technique. Vous trouverez des réponses à toutes les questions que vous êtes en droit de vous poser quant au fonctionnement et la structure interne des outils avec lesquels vous avez pu déjà vous familiariser.

Voyons d'abord comment est implanté géographiquement FORTH en mémoire (cf figure).



Le premier bloc, baptisé interpréteur, contient la partie "vivante" de FORTH. C'est une minuscule partie de code machine purement exécutable, dans laquelle vous êtes la plupart du temps : c'est elle qui assure l'enchaînement de tous les mots FORTH. Nous l'aborderons plus en détails par la suite.
Le second bloc est le dictionnaire. Au chargement de FORTH, il contient le FORTH de base. Il s'étend dans la mémoire au fur et à mesure des nouvelles définitions.
La DATA STACK nous est déjà familière : elle se propage en direction du dictionnaire. Son accès est réglementé de par sa structure de pile. Son adresse de base est dans la USER VARIABLE S0, et son sommet (top of stack) est accessible grâce au mot SP @ qui place au sommet de la pile l'adresse du sommet de la pile (!!).

La partie suivante est réservée au BUFFER de clavier de votre terminal : le TERMINAL INPUT BUFFER.

C'est là que FORTH range tous les caractères frappés au clavier. Il ne s'intéressera à son contenu que dans deux cas bien précis :

— soit si le BUFFER est plein,
— soit si vous avez frappé RETURN.

Ce buffer de taille fixe, est pointé indirectement par S0.

Plus bas dans la mémoire, on trouve un bloc de taille variable, appelé RETURN STACK.

C'est là que FORTH empile les adresses successives de retour des procédures.

L'adresse de base est contenue dans la USER VARIABLE R0 et son sommet est accessible grâce au mot RP @.

Le bloc suivant comprend toutes les valeurs des USER VARIABLES. Les mots qui permettent d'accéder à des valeurs sont bien sûr installés dans le dictionnaire.

Le dernier bloc est réservé aux BUFFERS d'E/S. C'est là que l'EDITEUR va ranger les pages que vous appelez.

[illegible] Description du dictionnaire

Le dictionnaire n'est autre qu'une liste de mots. Chaque nouveau mot greffé sur la liste a un lien logique avec le dernier défini.

Pour démarrer le processus, on postule que la liste contient au départ un élément virtuel "VIDE".

L'addition d'un mot nouveau dans le dictionnaire consiste dans la suite d'opérations suivante :

— Lier logiquement le nouveau mot au dernier mot entré.

— Marquer le nouveau mot comme dernier mot entré.

Voici comment ce mécanisme est réalisé pratiquement :

— Le lien logique est par adresses (LINK FIELD ADDRESS).

— L'élément "VIDE" a pour adresse 0. (LFA=0).

— L'adresse du LFA du dernier mot défini est obtenu par l'intermédiaire d'une USER VARIABLE.

— Bien que rien ne l'oblige, les mots sont implantés de façon séquentielle pour simplifier la gestion de la mémoire : Il faudra donc également connaître le premier emplacement disponible en mémoire (DP).

L'intérêt de cette structure est qu'elle est non-injective : Plusieurs mots de la liste peuvent pointer sur la même adresse, et donc avoir le même antécédent.

Il est donc possible de ramifier logiquement le dictionnaire en vocabulaires avec un seul jeu de pointeurs. (voir § II.3. et II.4).

A titre d'exercice, écrivons un mot FORTH qui compte le nombre de définitions actuellement présentes dans le vocabulaire courant.

```
: COMPTE
        0 CONTEXT @          ( Nb—el pointeur ——— )
        BEGIN
              @ PFA 4 — DUP
        WHILE
            SWAP 1+ SWAP
        REPEAT
        DROP
; [Return] OK
```

Dans le chapitre II, nous avons présenté le mot FORGET, qui "efface" du dictionnaire tous les mots définis depuis (et y compris) un mot donné.

On comprend mieux maintenant son mécanisme :

Il met dans DP l'adresse du mot qui précède le mot donné dans le dictionnaire.

Cet élagage du dictionnaire est brutal. Toute recherche dans le dictionnaire commence par le mot pointé par DP. Tous les mots définis postérieurement sont perdus.

Toutefois, FORGET est sécurisé grâce à la USER VARIABLE FENCE, qui donne une adresse limite au-delà de laquelle les suppressions sont rejettées.

A l'initialisation, FENCE contient l'adresse du dernier mot du FORTH de base.

Ainsi par exemple, vous pouvez à tout instant protéger tout votre travail d'un effacement intempestif en faisant :

```
DP @  FENCE ! [Return] OK
```

V.1.3. Observation d'un mot dans le dictionnaire

Voyons maintenant la structure d'un mot dans le dictionnaire.

Tout mot contient deux parties, un en-tête qui permet de l'identifier et un corps qui le définit.

a) en-tête

Le premier octet contient deux informations : la longueur du nom du mot sur les cinq bits de poids faible, et un état du mot sur deux bits.

7	6	5	4 3 2 1 0
1	P	S	LONGUEUR
Nom du mot			
LINK FIELD			

L'utilisation du SMUDGE BIT (S) a été vue dans le chapitre sur l'ASSEMBLEUR.

Il est mis 1 lorsque la définition du mot est valide.

Le PRECEDENTE BIT (P) est quant à lui lié au concept de mot immédiat, qui sera exposé dans un prochain chapitre.

Ensuite viennent les caractères ASCII du nom du mot (0 à 31 caractères), le MSB du dernier octet est positionné à 1 pour signifier la fin de la chaîne.

L'adresse du premier octet de l'en-tête est par convention appelée NFA (NAME FIELD ADDRESS).

On trouve ensuite la zone qui assure le chaînage dans le dictionnaire.

Elle contient le NFA du mot précédent dans le dictionnaire :

C'est le LFA (LINK FIELD ADDRESS).

Le contenu du LFA du premier mot du dictionnaire est bien sûr nul.

b) Le corps

L'adresse de début de ce bloc est appelé CFA (CODE FIELD ADDRESS).

C'est là que commencent toutes les données relatives à la fonction du mot.

Ce CODE FIELD peut contenir trois sortes d'informations :

— soit du code exécutable (mot contenant de l'ASSEMBLEUR)

— soit des paramètres (cas des variables)

— soit enfin dans le cas le plus courant, la liste des CFA des mots qui le constituent

On distingue artificiellement une adresse supplémentaire, le PFA (PARAMETER FIELD ADDRESS), car FORTH intercale entre l'en-tête et les paramètres du mot une information sur deux octets dont nous exposerons bientôt l'utilité. Il y a donc toujours une différence de deux entre PFA et CFA.

Il existe un certain nombre de mots qui permettent d'accéder à ces différentes adresses. Tous sont construits autour du mot ' (prononcez tick), qui retourne sur la pile de PFA d'un mot donné.

Exemple:

```
HEX [Return] OK
' SWAP . [Return] 054B OK
```

On construit alors facilement CFA, LFA et NFA de la façon suivante :

```
: CFA   2   -;  [Return] OK ( PFA --- CFA )
: LFA   2   -;  [Return] OK ( CFA --- LFA )
: NFA                       ( CFA --- NFA )
      4 -
      BEGIN
            DUP
            80  AND  0=
      WHILE
            1  -
      REPEAT
 ; [Return] OK
```

NFA explore les octets vers les adresses décroissantes, en cherchant le premier dont le MSB est à 1.

Revenons plus en détail sur le contenu du CFA d'un mot quelconque.

Le contenu de cette adresse pointe sur un mot dit de définition, mot qui dépendra de la nature du mot choisi.

Si le mot étudié est une variable, l'exécution du mot de définition correspondant, dont l'adresse se trouve donc dans le CFA, placera sur la pile le PFA, à savoir l'adresse où l'on trouvera la valeur de la variable.

Si le mot étudié est une constante, l'exécution du mot de définition correspondant placera cette fois sur la pile le contenu de la cellule mémoire d'adresse PFA, à savoir la valeur de la constante.

Si le mot étudié est un mot classique, c'est-à-dire dont la définition commence par le mot : , l'exécution du mot de définition enchaînera l'exécution des mots qui composent la définition.

Comment sont représentés ces mots à l'intérieur de la défintion ?

Tout simplement par leur CFA respectif.

Exemple:

```
10 VARIABLE POIDS [Return] OK
```


Le mot POIDS aura donc dans le dictionnaire la structure suivante :

NFA	85	P
	O	I
	D	S+80
LFA	NFA du mot précédent.	
CFA	Adrèsse mot de définition	
PFA	Réservé	
PFA+2	Valeur de la variable.	

La structure d'une constante sera absolument similaire, seul diffère le mot de définition pointé.

Prenons maintenant le cas d'un mot classique :

```
: REPAS ENTREE PLAT DESSERT ; [Return] OK
```

Le mot REPAS aura la structure suivante :

NFA	85	R
	E	P
	A	S+80
LFA	Ad. mot précédent	
CFA	Ad. mot de def.	
	CFA de ENTREE	
	CFA de PLAT	
	CFA de DESSERT	
	CFA de ;	

La structure du mot dans le dictionnaire étant maintenant connue, nous allons pouvoir nous intéresser au fonctionnement de l'interpréteur.

V.1.4. Fonctionnement interne de l'interpréteur

L'interpréteur est alimenté mot par mot (Input Stream).

Il peut choisir deux traitements pour les mots qui lui sont présentés : Compilation ou Exécution. Ce choix est déterminé par l'état d'une USER VARIABLE : STATE.

V.1.4.1. Compilation

Toute définition doit commencer par le mot : .

On devine que c'est lui qui va faire passer l'interpréteur en mode compilation par l'intermédiaire de STATE, après avoir constitué la tête de chaîne dans le dictionnaire.

Tous les mots qui suivent dans l'input stream sont compilés : leur CFA est ajouté systématiquement à la position courante du pointeur de dictionnaire.

Vous savez aussi que c'est le mot ; qui termine toute définition : c'est lui qui remet l'interpréteur en mode exécution.

Vous êtes en droit de vous poser la question suivante :

Si l'interpréteur, après la rencontre du mot : , compile systématiquement tous les mots qui suivent dans l'input stream, et que le mot ; est un mot FORTH au même titre que les autres, comment peut-il avoir une action quelconque sur l'état de l'interpréteur ?

En fait, le mot ; fait aussi partie de cette classe spéciale de mots dits de définition, qui peuvent avoir une fonction différente selon l'état où se trouve l'interpréteur lorsqu'il les rencontre. Nous verrons ces notions dans le chapitre VI.

La fonction compilation de l'interpréteur consiste donc à ajouter en fin de dictionnaire le CFA du mot courant de l'input stream. Cette fonction élémentaire est réalisée par le mot COMPILE.

Notons que cette notion d'input stream est large, les mots pouvant être entrés à partir du clavier, de pages de l'éditeur (lors du LOAD), ou de tout type d'interface.

Le travail préliminaire consiste à reconnaître le mot dans le dictionnaire, pour obtenir son CFA. Cette fonction est analogue à celle du mot , .

Deux cas peuvent se présenter :

— *le mot existe* : pas de problème.

— *le mot n'existe pas*, auquel cas, l'interpréteur va chercher si son

expression est compatible avec un nombre de la base courante. Avant de le placer dans le dictionnaire, il y intercalera le CFA du mot spécialisé LITERAL qui, lors de l'exécution, évitera la confusion de ce nombre avec un CFA à exécuter, et le placera au sommet de la pile.

Le mécanisme qui construit une définition à partir de mots présents dans le dictionnaire étant maintenant éclairci, voyons comment ces mots sont exécutés.

V.1.4.2. Exécution

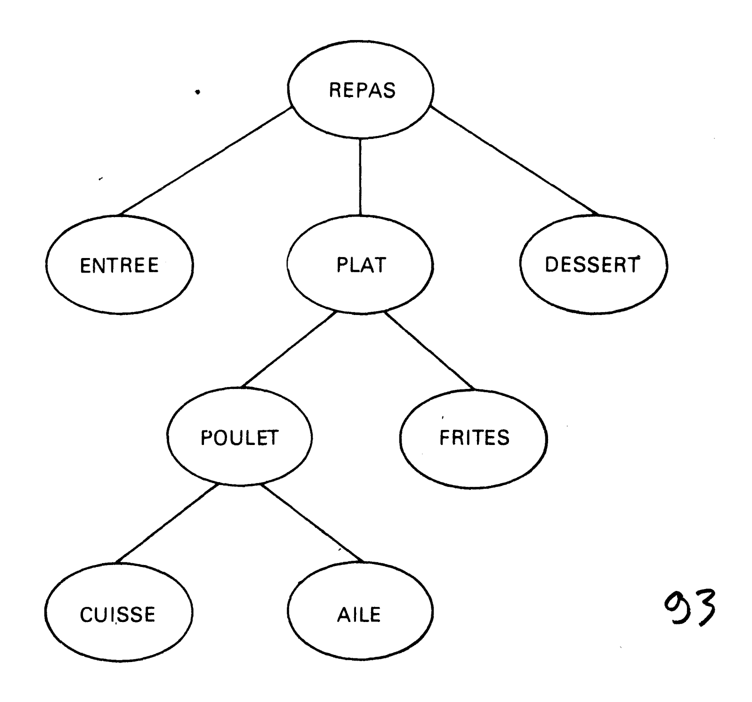
Que se passe-t-il lorsque vous tapez un mot FORTH sur le clavier de votre terminal et que vous lancez son exécution en pressant [Return] ?

L'interpréteur, qui est alors en mode exécution, placera le CFA de ce mot au sommet de la pile, et sous-traitera le travail au mot EXECUTE.

Pour mieux vous représenter son mécanisme, nous allons revenir à des bases plus théoriques, en l'occurence le problème du parcourt d'un arbre.

Soit un repas composé d'une entrée, d'un plat, et d'un dessert. Le plat lui-même comprend du poulet et des frites. Il y a deux morceaux de poulet, la cuisse et l'aile.

Il est naturel de représenter le repas comme suit :

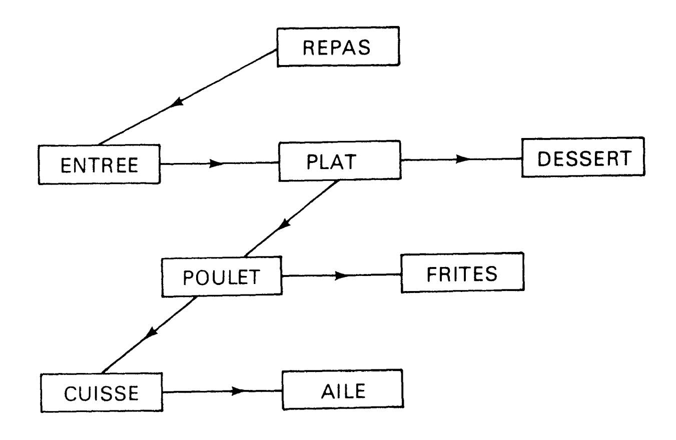


Le pr[illegible]ne [illegible] co[illegible]ire [illegible]éca[illegible] qu[illegible]amè[illegible]us le[illegible] constituants du repas dans l'ordre où ils vont être mangés c'est-à-dire ENTREE, CUISSE, AILE, FRITES, DESSERT.

Il faut parcourir l'arbre d'une façon cohérente en ne sélectionnant que les feuilles.

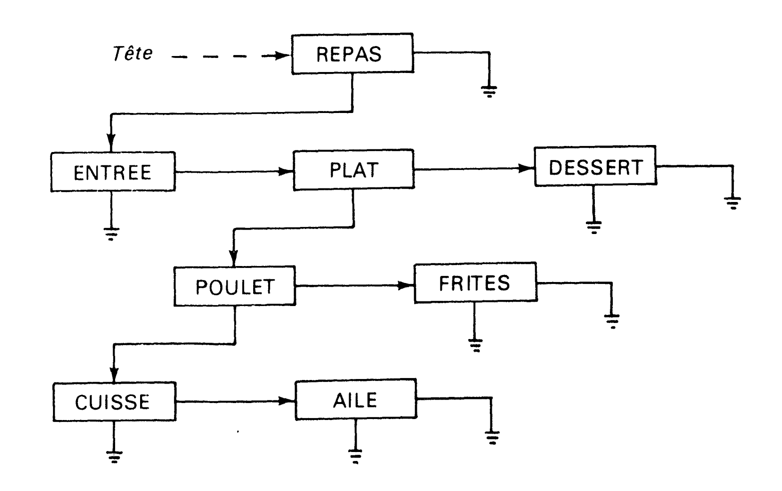
Tel qu'il est figuré ci-dessus, l'arbre est peu pratique d'emploi, car il manque une relation d'ordre entre les nœuds d'un même niveau.

Il est beaucoup plus seyant d'adopter la représentation suivante :



Il est nécessaire en outre de savoir par où commencer, c'est-à-dire de connaître la tête de l'arbre.

Pour le rendre encore plus exploitable par un mécanisme informatique, voici une représentation un peu plus élaborée :



On convient que l'accès à l'une des cases fournit les trois informations suivantes :

— le contenu de la case,

— l'adresse de la case "fille" (en-dessous),

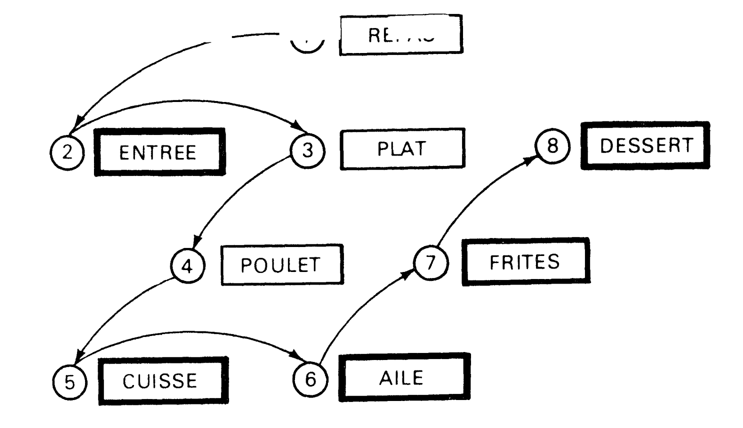
— l'adresse de la case "sœur" (à droite).

Ces deux adresses prenant conventionnellement la valeur 0 (vide) lorsque cette "fille" ou cette "sœur" n'existent pas.

Le mécanisme de parcours est le suivant :

```
EMPILER tête
REPETER
      DEPILER adresse
      TANT QUE adresse ≠ 0 FAIRE
          SI adresse (fille) = 0 ALORS
                MANGER contenu
                adresse = adresse (sœur)
          SINON
                EMPILER adresse (sœur)
                adresse = adresse (fille)
          FIN SI
      FIN TANT QUE
JUSQU'A pile = VIDE
```

Le sens du parcours est alors :



Les plats consommés étant ceux encadrés en gras.

Que faut-il retenir de cet exemple:

— Toute exploration de structure arborescente nécessite une pile auxiliaire car il faut, après avoir parcouru une branche donnée, pouvoir revenir à sa jonction avec l'arbre de façon à en pouvoir explorer une nouvelle. Ces jonctions sont utilisées dans l'ordre inverse de leur rencontre: cela correspond parfaitement à la structure de pile LIFO.

— Il faut disposer de marqueurs de fin d'arborescence pour déclencher le dépilage de la jonction précédente.

Vous avez certainement deviné qu'il existait un lien direct entre ce problème et l'exécution du mot FORTH suivant.

: REPAS ENTREE PLAT DESSERT ; Return OK

Avec:

: PLAT POULET FRITE ; Return OK

: POULET CUISSE AILE ; Return OK

On peut distinguer trois fonctions différentes dans l'algorithme:

— *les tests:* les définitions sont terminées (marquées) par le mot ; qui termine les niveaux horizontaux.

Le test d'existence d'un niveau inférieur n'a pas lieu d'être car les feuilles sont composées d'assembleur, lequel rétablit de lui-même la logique de parcours de l'ensemble.

— *gestion de la pile:* l'empilage est assuré par le mot : de même le dépilage est assuré par le mot ; . Ces deux mots font partie d'une classe un peu particulière de mots de haut niveau: ce sont les mots de définition dont nous parlerons dans le chapitre suivant.

— *gestion des adresses:* Cette fonction, extrêmement simple, est réalisée par un module assembleur de quelques lignes, qui est la seule partie de FORTH qui ne s'intègre pas dans la structure de dictionnaire.

La grande particularité du langage FORTH est que le dictionnaire se suffit à lui-même puisque chaque mot contient la logique d'enchaînement des exécutions. C'est un des principes de fonctionnement des langages dits tissés.

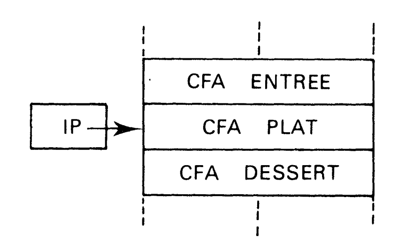
Pour vous en convaincre, suivons pas à pas le déroulement du mot REPAS, juste après l'exécution d'ENTREE.

Les registres dont l'interpréteur FORTH a besoin en mode exécution sont:

RP: pointeur de pile de retour.

IP : pointeur d'interprétation. Au cours de l'exécution d'une définition, il pointe en permanence sur la cellule mémoire contenant le CFA du prochain mot à exécuter.

Comment se déroule l'exécution de PLAT?

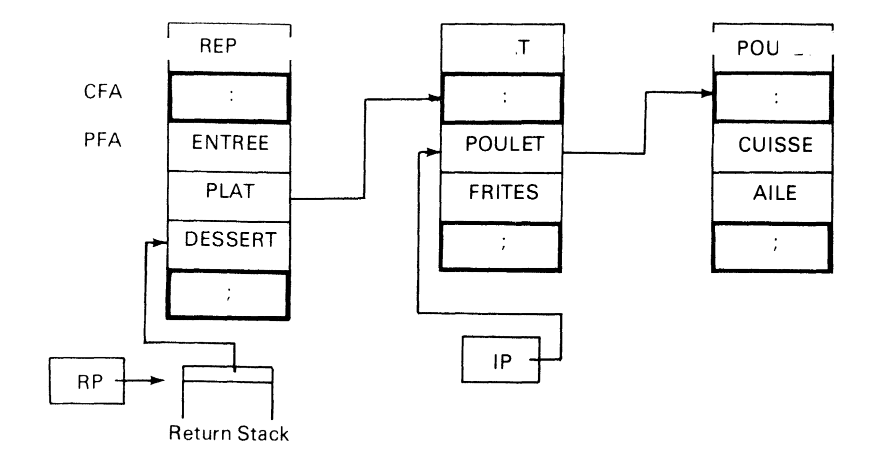


IP pointe sur le CFA de plat. Avant d'exécuter ce mot, on incrémente IP de 2 pour le faire pointer sur la cellule contenant le CFA de DESSERT.

En effet, c'est le mot qu'il nous faudra exécuter lorsque nous aurons fini notre plat.

Pour consommer notre plat, il nous faut dérouler le code pointé par le CFA de plat.

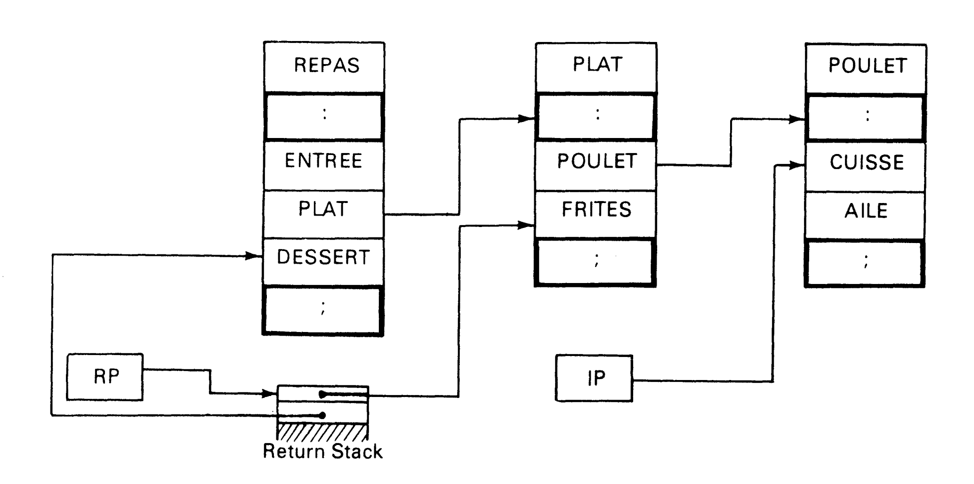
Le premier mot rencontré va sauvegarder IP dans la pile de RETURN, et met dans IP le PFA de PLAT. IP pointe donc maintenant sur une cellule contenant le CFA de POULET.



Comme nous l'avons fait avant l'exécution de PLAT, nous allons incrémenter IP afin qu'il pointe sur FRITES à notre retour de POULET.

Le premier mot rencontré dans la consommation POULET sauve IP sur la Return Stack et le remplace par le PFA de POULET, cellule contenant le CFA de CUISSE.

On a alors les chaînages suivants:



Projetons nous en avant dans le temps, et considérons que nous en avons fini avec la CUISSE et l'AILE. Nous allons donc exécuter le ; de POULET, placé dans dictionnaire en fin de définition, lors de la compilation de POULET.

Comme d'habitude, avant d'exécuter le mot ; , l'interpréteur incrémente IP, ce qui pourrait paraître inquiétant puisque la définition s'arrête là.

L'exécution du ; remet cependant tout en ordre, puisqu'elle remplace la valeur courante de IP par le sommet de la pile de Return, qu'elle dépile.

IP pointe donc maintenant sur le CFA de FRITES, c'est bien la suite de notre menu.

De même, l'exécution du ; de PLAT remplacera IP par le contenu du sommet de la pile, qui n'est autre que le CFA de DESSERT.

V.1.5. Processus d'initialisation

Il y a trois types d'initialisation: Les piles de données et de return, les I/0 BUFFERS, les USER VARIABLES.

Trois mots réalisent des initialisations: on peut résumer leurs actions respectives dans le tableau suivant:

	DATA & RETURN STACK	IOBUFFERS	USER VARIABLES
COLD	oui	oui	oui
WARM	oui	oui	non
ABORT	oui	non	non

Les piles:

L'initialisation des piles se fait en mettant la valeur des USER VARIABLES S0 et R0 dans les pointeurs de pile SP et RP.

Les mots qui réalisent ces fonctions sont SP! et RP!.

On a donc de façon très simple:

`: ABORT SP! RP! QUIT ;` Return OK

Il faut noter que l'action de ABORT dépend du contenu des USER VARIABLES R0 et S0 au moment où il est exécuté.

QUIT permet d'entrer dans FORTH. Voyez plus en détail le paragraphe suivant.

Les IOBUFFERS:

Le mot EMPTY-BUFFERS prend en charge toute l'initialisation des

USER VARIABLES poi sur buf

: WARM EMPTY-BUFFERS ABORT ; Return OK

Les USER VARIABLES:

Plutôt que de les initialiser séparément, il est plus simple de garder une copie des valeurs initiales dans une partie de la mémoire, pour en faire une recopie complète sur la zone de travail.

La définition de COLD sera donc :

```
: COLD
        Adresse début zone
        Adresse fin zone
        Nb octets
        CMOVE
        WARM
; Return OK
```

V.1.6. Le moniteur FORTH

Dans la présentation que nous venons de vous faire du déroulement de FORTH, il manque un dernier élément qui réalise la jonction entre vous et l'interpréteur.

C'est la fonction QUIT.

Comme sur tout système, vous êtes en permanence dans une boucle en attente de commandes.

On peut décomposer les fonctions de QUIT de la façon suivante :

— Initialisation de la RETURN STACK (RP!).
— Attente d'un déclenchement d'un traitement (touche Return).
— Interprétation.
— Affichage de OK

Le mot QUIT peut bien sûr être inclus dans une définition : il vous replace dans l'état initial. C'est pourquoi le premier mot de QUIT réalise l'initialisation de la Return Stack.

L'enchaînement général de FORTH se résume donc comme suit :

V.2. IMPLANTATION SUR DIFFÉRENTES MACHINES

FORTH a besoin de trois pointeurs pour fonctionner :

RP : pointeur de pile de retour
SP : pointeur de pile de données
IP : pointeur d'interprétation

Généralement, les micro-processeurs savent gérer automatiquement une pile pointée par un registre spécialisé du CPU, accessible à l'utilisateur.

Cette pile est utilisée par la CPU pour sauvegarder les valeurs du compteur ordinal lors des appels de sous-programmes. On dispose également des instructions d'accès type PUSH, et POP qui réalisent l'accès mémoire et la mise à jour du pointeur de pile en une instruction élémentaire.

On utilise souvent les facilités de cette gestion automatique pour implanter la pile de données.

Si vous ne disposez pas d'une seconde pile automatique, vous devrez assurer une gestion manuelle de la pile des return.

Deux cas peuvent se présenter : soit vous disposez de suffisamment de registres libres capables de contenir des adresses, auquel cas vous en dédiez un à la gestion de la pile des return. Si tel n'est pas le cas, vous gardez ce pointeur en mémoire à une adresse fixe.

[illegible]ur I[illegible] prol[illegible] est [illegible]aire [illegible] lui [illegible]er u[illegible] registr[illegible] [illegible]adres-sage dans la CPU, soit l'installer en mémoire.

Si le nombre des registres disponibles dans la CPU est limité (Z 80 par exemple), vous devrez faire, pour l'implantation des trois pointeurs, un compromis entre la mémoire et les registres internes.

Il faut bien voir que ce choix dépendra essentiellement des facilités de la machine.

Pour vous en convaincre, voici deux exemples réels:

Le 8080, microprocesseur 8 bits bien connu, dispose des registres suivants: A , F , B , C , D , E , H , L , PC , SP.

. SP est le pointeur de pile géré automatiquement par la machine lors des instructions de type PUSH, POP, CALL.

. PC est le compteur ordinal.

. (B,C), (D,E), (H,L), trois paires de registres accessibles soit individuellement, soit par deux pour manipuler les adresses.

. A l'accumulateur et F le registre d'état de l'accumulateur.

Une des particularités de ce micro-processeur est que seul HL peut adresser la mémoire, on note d'ailleurs conventionnellement M = (HL).

On prendra naturellement SP pour gérer la pile de données.

Il reste trois registres machine (BC, DE, HL) pour les deux registres FORTH RP et IP.

La programmation en assembleur des mots FORTH sera absolument inextricable si deux doubles registres sur trois doivent être conservés intacts.

La sagesse consiste donc à attribuer BC à IP par exemple et à garder RP en mémoire.

Le choix de RP et IP est indifférent car leurs fonctions sont étroitement liées et leurs sollicitations égales.

Sur d'autres machines, l'architecture interne fait que les registres internes ne sont pas privilégiés par rapport à la mémoire, grâce à des modes d'adressage très évolués (LSI11, HP3000....).

Il n'y a aucune difficulté à mettre tous les pointeurs en mémoire, leur gestion restant très simple. Les paramètres à prendre en compte pour la répartition sont alors le temps d'exécution, la souplesse de programmation et le respect des contraintes du système d'exploitation.

La deuxième étape consiste à dimensionner les différents blocs qui composent FORTH en mémoire.

Les IOBUFFERS: leur taille est déterminante du nombre de pages résidentes simultanément en mémoire.

La RETURN STACK : Bien évidemment, sa taille limite le nombre d'imbrications maximum admissible dans une définition.

TERMINAL INPUT BUFFER : Il est souhaitable de le faire coïncider avec la longueur d'une ligne sur votre terminal.

DATA STACK : Elle se partage, avec le dictionnaire, tout le reste de la mémoire.

Ces paramètres étant définis, il vous reste à écrire en assembleur le source, composé lui-même de 10 % de code exécutable, le reste n'étant que des références à des étiquettes déjà définies.

En d'autres termes, l'adaptation du FORTH de base d'une machine sur une autre ne nécessite que la traduction de ces 10 % de code purement exécutable.

Mot	*Syntaxe*	*Définition*
'	' Mot (--- Pfa)	Place sur la pile le Pfa du mot qui suit la commande dans l'input stream.
PFA	PFA (Cfa --- Pfa)	Calcule le Pfa à partir du Cfa.
LFA	LFA (Cfa --- Lfa)	Calcule le Lfa à partir du Cfa.
NFA	NFA (Cfa --- Nfa)	Calcule le Nfa à partir du Cfa.
CFA	CFA (Pfa --- Cfa)	Calcule le Cfa à partir du Pfa.
EXECUTE	EXECUTE (Cfa ---)	Exécute le mot dont le Cfa est sur la pile.
COLD	COLD	Initialisation des piles, des IObuffers, et des User Variables.
WARM	WARM	Initialisation des piles et des IObuffers.
ABORT	ABORT	Initialisation des piles.
SP!	SP!	Met dans le pointeur de pile la valeur de la User Variable S0.
SP@	SP@	Charge sur la pile la valeur du pointeur de pile
RP!	RP!	Met dans le pointeur de la pile de retour la valeur de la User Variable R0.
RP@	RP@	Charge sur la pile la valeur du pointeur de la pile de retour.
QUIT	QUIT	Initialise la pile de retour, arrête la compilation ou l'exécution, et rend la main à l'utilisateur.

VI

LES MOTS DE HAUT NIVEAU

VI.1. LES MOTS AYANT TRAIT A LA COMPILATION

VI.1.1. Concept de mot immédiat et non-immédiat

Dans le chapitre précédent, nous avons évoqué la possibilité pour certains mots d'avoir un comportement différent selon que l'interpréteur est en mode compilation ou en mode exécution.

L'intérêt de cette possibilité est de pouvoir exercer une action pendant la construction d'une définition.

Pour bien vous fixer les idées, voyons d'abord la catégorie de mots dont l'action ne se situe qu'au moment d'une compilation : il s'agit des mots immédiats.

La génération de tels mots est très simple : le mot IMMEDIATE rend immédiat le dernier mot entré dans le dictionnaire.

Prenons tout de suite un exemple :

```
: COUCOU ."  COUCOU" ; [Return] OK
IMMEDIATE [Return] OK
: TEST COUCOU ."  COCORICO" ; [Return] COUCOU OK
TEST [Return] COCORICO OK
COUCOU [Return] COUCOU OK
```

[illegible] co[illegible]nd [illegible] cet e[illegible]le q[illegible]ès a[illegible] rend[illegible]tre [illegible]mmé-
diat, s'il est présent par la suite dans une définition, il sera exécuté à la compilation sans être pris en compte dans cette définition.

Par ailleurs, la propriété pour un mot d'être immédiat n'affecte en aucune manière son comportement lorsqu'il est directement exécuté.

L'intérêt de pouvoir agir au moment de la compilation ne se limite bien sûr pas à l'affichage de niaiseries sur votre terminal.

Essayons d'imaginer comment sont implantées les structures de contrôle de FORTH. Vous disposez au départ des mots FORTH BRANCH et 0BRANCH qui sont des GOTO extrêmement rudimentaires : ils exécutent un adressage relatif par rapport à leur position dans la définition.

```
: -4 -4 ; IMMEDIATE [Return] OK
: PARTOUT ." PARTOUT" CR ; [Return] OK
: ZOU PARTOUT BRANCH -4 ; [Return] OK
```

0BRANCH quant à lui fera un branchement relatif à condition que la valeur présente au sommet de la pile soit nulle.

C'est intentionnellement que nous ne détaillerons pas plus ce type de mots qui sont à proscrire dans vos définitions.

Prenons le cas très simple de BEGIN---AGAIN. A l'exécution, il suffit en fait que AGAIN exécute un BRANCH à "l'étiquette" BEGIN. On comprend bien que BEGIN n'a rien à exécuter, il sert au moment de la compilation de point de repère pour la compilation de AGAIN. Son rôle est bien immédiat, il prépare le travail pour la compilation de AGAIN, mais ne figurera plus dans la définition compilée.

Comme toujours, le passage de paramètre se faisant par l'intermédiaire de la pile, BEGIN va empiler l'adresse de la première cellule libre dans le dictionnaire.

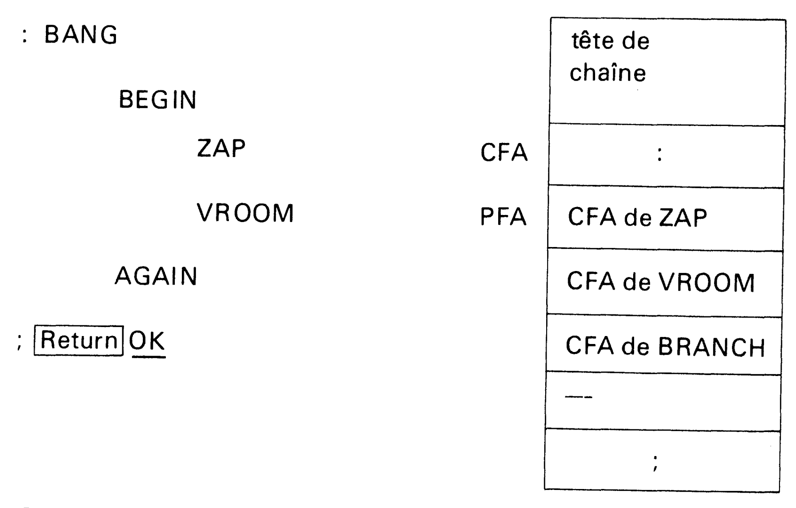
```
: BEGIN HERE ; IMMEDIATE [Return] OK
```

AGAIN, quant à lui, va ranger le CFA de BRANCH dans la première cellule libre puis l'adresse relative du branchement, à savoir la différence entre HERE et l'adresse placée au sommet de la pile par BEGIN.

AGAIN est donc défini de la façon suivante :


```
: [illegible]IN C[illegible]ILE [illegible]ICH    ([illegible]ace l[illegible] de [illegible]NCH [illegible] suite
                                du dictionnaire)
  HERE  -                       (place l'adresse relative du branchement)
  ;
  IMMEDIATE [Return] OK
```

Observons la définition d'un mot utilisant cette structure de contrôle :



Comment fait l'interpréteur pour reconnaître un mot immédiat ? C'est le PRECEDENCE BIT qui indique le caractère immédiat d'un mot. La fonction du mot IMMEDIATE est de positionner le precedence bit du dernier mot créé dans le dictionnaire.

Étant donné que, lorsque l'interpréteur est en mode compilation et qu'il rencontre un mot immédiat il l'exécute, il est bon de se demander ce qui fait que l'interpréteur se retrouve dans ce même mode pour le reste de la définition. En fait l'interpréteur reste toujours en mode compilation lorsqu'il compile une définition, même pendant l'exécution d'un mot immédiat rencontré.

Lorsque l'interpréteur détecte un mot immédiat dans une définition, il effectue un EXECUTE de ce mot, ce qui ne nécessite aucun changement de mode.

VI.1.2. Le mot [COMPILE]

Supposez que vous avez défini un mot et que vous l'avez rendu immédiat. Il est possible que vous ayez ensuite besoin de vous en servir dans une

dé...on s... sa fo... norm... c'est ... ...re n... imméd... Pou... ...oud ce problème, il vous suffira de précéder ce mot dans la définition par le mot [COMPILE]. Ce mot [COMPILE] indique à l'interpréteur qu'il ne doit pas tenir compte du precedence bit du mot qui suit.

Exemple 1 :

Dans certains FORTH, le mot ' (tick) est immédiat. Si vous avez besoin de travailler sur le PFA d'un mot donné à l'exécution d'une définition, il est nécessaire de précéder ' par [COMPILE].

Supposons que l'on veuille obtenir le CFA d'un mot en utilisant la syntaxe : CFA mot.

Si l'on utilise ' sous sa forme immédiate sans [COMPILE], c'est-à-dire :

```
: CFA ' 2 -; Return OK
```

Le fait de taper CFA mot aurait pour effet de mettre le CFA de 2– sur la pile et d'exécuter le mot, ce qui n'est pas le but recherché.

La solution est d'inhiber l'action de ' au moment de la compilation en la déportant au moment de l'exécution, d'où la définition de CFA :

```
: CFA [COMPILE] ' 2 - ;        ( ----- CFA )
Return OK
```

Exemple 2 :

Une autre application est de se créer un pseudo-vocabulaire des mots immédiats, c'est-à-dire de pouvoir définir des mots immédiats à partir d'autres mots immédiats.

```
: BOUM ." BOUM" ; IMMEDIATE Return OK
: BADABOUM ." BADA" [COMPILE] BOUM ; IMMEDIATE
Return OK
```

VI.1.3. La structure [-----]

Nous allons maintenant nous intéresser à la structure qui va nous permettre de rendre immédiat un ou une série de mots à l'intérieur d'une définition de manière temporaire.

Le mot [signale le début des mots à considérer comme immédiats, et] le retour à la compilation normale.

L... les ... pla... entre [et] gardent le même caractère non-immédiat pour toute utilisation future.

Exemple :

```
: VLAN ." VLAN" ; Return OK
: PAF [ VLAN ] VLAN ; Return VLAN OK
PAF Return VLAN OK
: PIF VLAN ; Return OK
```

De même que [COMPILE] rend non-immédiat le mot immédiat qui le suit dans une définition, sans pour autant en affecter la nature, les mots [et] n'ont pas rendu VLAN immédiat. C'est à l'utilisateur de déterminer sous quelle forme un mot sera utilisé le plus souvent avant de rendre un mot immédiat, de façon à ne pas alourdir inutilement les définitions futures.

VI.1.4. COMPILE–TIME et RUN–TIME

Nous avons vu qu'il existe deux modes pour l'interpréteur : le mode compilation et le mode exécution.

Nous savons définir deux types exclusifs de mots : les mots non-immédiats, dont l'action se déroule lorsque l'interpréteur est en mode exécution et les mots immédiats, qui sont exécutés lorsque l'interpréteur est en mode compilation.

Les structures de contrôle ne rentrent cependant pas dans ce cadre car la plupart des mots qui les composent nécessitent deux actions différentes, l'une à la compilation et l'autre à l'exécution. C'est le cas de AGAIN :

— *Compilation :* il calcule et mémorise l'adresse relative du BEGIN dont l'adresse absolue se trouve au sommet de la pile.

— *Exécution :* il effectue le branchement à l'adresse du BEGIN.

Essayons de dégager une méthodologie de résolution de ce type de problème.

Il est évident que le mot doit agir à la compilation ; il doit donc être immédiat. Ainsi, tous les mots qui le composent seront exécutés lors de la compilation. A priori il n'en restera aucune trace dans le mot en cours de définition, sauf si l'exécution du mot immédiat force dans la définition les mots remplissant la deuxième fonction du mot immédiat.

Il existe trois mots (déjà définis) qui permettent de forcer des objets dans le dictionnaire : COMPILE, C, et , .

COMPILE place le CFA du mot qui le suit à l'adresse contenue dans HERE, c'est-à-dire à la fin du dictionnaire.

C, et , placent respectivement le sommet de la pile au même endroit (HERE) sur respectivement un ou deux octets.

Nous vous rappelons la définition de AGAIN qui représente un exemple simple :

Sachant que BEGIN a placé son adresse absolue sur la pile,

```
: AGAIN
        COMPILE   BRANCH  (force le CFA de BRANCH)
        HERE  -  ,        (calcul de l'adresse relative)
; IMMEDIATE [Return] OK
```

Exemple 1 :

```
: SPLATCH ." SPLATCH" ; [Return] OK
: PLOF ." PLOF" ; [Return] OK
: BRUIT COMPILE PLOF SPLATCH ; IMMEDIATE [Return] OK
: SON BRUIT ; [Return] SPLATCH OK
SON [Return] PLOF OK
```

Exemple 2 :

Essayons de définir les mots FORTH de la structure de contrôle DO——LOOP.

Analysons les fonctions des deux mots de la structure.

DO : — *Compilation* : il empile son adresse dans la définition.
— *Exécution* : il transfère les indices de la boucle sur la Return Stack.

LOOP : — *Compilation* : il calcule l'adresse relative de DO.
— *Exécution* : il incrémente l'indice de boucle, effectue le test et éventuellement le saut.


```
: DO
    COMPILE  SWAP
                      (transfert des indices de
    COMPILE >R         boucle vers la Return Stack
    COMPILE >R
    HERE              (empile l'adresse absolue)
; IMMEDIATE [Return] OK

: LOOP
    COMPILE R>        (incrémente l'indice courant de boucle)
    COMPILE 1+
    COMPILE R>        (empile l'adresse de fin)
    COMPILE 2DUP
                      (test de fin de boucle
    COMPILE >R         et sauvegarde des
                       deux indices de boucle)
    COMPILE >R
    COMPILE <
    COMPILE 0=
    COMPILE           (branchement conditionnel)
    0BRANCH
    HERE  - ,         (force l'adresse du branchement)
    COMPILE R>
                      (nettoie la Return Stack)
    COMPILE R>
    COMPILE 2DROP
; IMMEDIATE [Return] OK
```

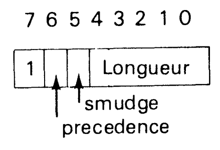
A titre d'exercice, définissez +LOOP et la structure BEGIN——WHILE——REPEAT.

Nous espérons que vous êtes conscients de la puissance des outils que FORTH vous offre avec la possibilité de créer des structures de contrôle. Si vous connaissez un autre langage évolué, vous en apprécierez d'autant plus FORTH.

VI.2. LES MOTS AVANT TRAIT A LA DÉFINITION D'AUTRES MOTS

VI.2.1. Le bit de validation de définition (SMUDGE BIT)

A plusieurs reprises déjà, nous avons évoqué deux bits spécialisés dans le premier octet des définitions contenues dans le dictionnaire.



Maintenant que vous avez une idée précise du fonctionnement interne de l'interpréteur, nous allons revenir de manière plus approfondie sur la fonction de ces différents bits.

— Bit 7 : Il vaut toujours 1. Pourquoi ?

Vous avez remarqué que vous ne pouvez accéder à une définition dans le dictionnaire que par son LFA. A l'instar du LFA, le NFA n'est pas immédiatement calculable, puisque les noms de définition sont de longueur variable.

Grâce à ce bit systématiquement forcé à 1 en tête de définition, nous allons pouvoir le retrouver de façon très simple : il suffit de rechercher vers les adresses décroissantes le premier octet dont le MSB est à 1.

```
: NFA 4 -              ( CFA----- )
    BEGIN
        DUP @          ( CFA (CFA)----- )
        80 AND 0=      ( CFA flag---- )
    WHILE
        1-             ( CFA-1----- )
    REPEAT
;[Return]OK
```


Tel que ce mot a été défini, vous pouvez écrire :

```
HEX ' NFA 2 - NFA[Return] 093A OK
```

093A est ici l'adresse de la tête de chaîne du mot NFA.

Si vous souhaitez une utilisation moins lourde, il vous sera facile de définir :

```
: NOM HEX
      [COMPILE] ' 2 -
      NFA DECIMAL
; IMMEDIATE[Return]OK
```

et vous aurez alors :

```
NOM NFA[Return]093A OK
```

— Bit 6 : C'est le précédence bit.

C'est lui qui spécifie à l'interpréteur le caractère immédiat du mot considéré.

Rappelons que IMMEDIATE force à 1 le precedence bit du dernier mot créé.

Il est facile de construire des mots capables de modifier le precedence bit de n'importe quel mot du dictionnaire.

```
: RESETPREC               ( CFA--- )
        [COMPILE] NOM     ( NFA--- )
        DUP C@            ( NFA (NFA)--- )
        BF AND
        SWAP              ( (NFA)' NFA--- )
        C!
; [Return]OK
```

```
: SETPREC            ( CFA --- )
        [COMPILE] NOM
        DUP C@
        40 OR
        SWAP
        C!
  ; [Return] OK
```

Il convient de noter la différence essentielle qui existe entre le mot [COMPILE] et RESETPREC : le premier n'affecte pas le precedence bit, alors que le second a été écrit pour le mettre à 0.

— Bit 5 : C'est le smudge bit.

Ce bit signale à l'interpréteur la validité de la définition.

Vous avez sans doute déjà été confronté au problème de définition non-valide.

C'est le cas lorsque l'interpréteur rejette la compilation d'une définition avec un message d'erreur.

```
: TEST BEGIN ; [Return] MSG#0 OK
```

Le mot est bien présent dans le dictionnaire comme le prouve VLIST, alors que toute tentative d'exécution se solde par un message d'erreur (mot inexistant).

Du fait de l'analyse séquentielle de la définition, : a bien créé une tête de chaîne correspondant à ce mot, que VLIST a su reconnaître. Par contre, l'interpréteur a détecté une anomalie de compilation et n'a donc pas positionné le smudge bit, ce qui empêche toute utilisation de cet embryon de définition.

Lorsque la définition est correcte, c'est ; qui est chargé de positionner ce bit. Si votre définition ne se termine pas par ; , lorsque vous utilisez ;CODE par exemple, il ne faudra pas oublier de le positionner immédiatement à l'aide de SMUDGE.

— Bits 4, 3, 2, 1, 0 : Ces cinq bits contiennent la longueur du nom de la définition.

VI.2.2. Les mots de définition connus

On appelle mot de définition tout mot capable de créer une tête de chaîne.

Jusqu'à maintenant, nous en avons rencontré quatre :
: , VARIABLE , CONSTANT et USER.

Tous les quatre créent une tête de chaîne de même structure, à savoir :

— Un premier octet tel que détaillé au paragraphe précédent.

— Une chaîne de caractères (codes ASCII) contenant le nom du mot, dont le MSB du dernier octet est forcé à 1.

— Un pointeur de chaînage vers le mot précédent du vocabulaire courant.

Pour aborder en douceur l'objet de ce chapitre, penchons nous ensemble sur le comportement du mot CONSTANT.

Vous savez comment définir une constante :

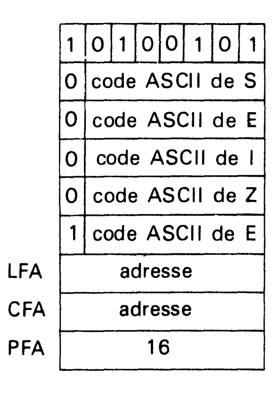
```
16 CONSTANT SEIZE [Return] OK
```

Puis relire sa valeur :

```
SEIZE . [Return] 16 OK
```

C'est manifestement l'exécution du mot SEIZE qui retourne la valeur de la constante sur la pile, sans qu'aucune compilation n'ait eu lieu par l'intermédiaire de : par exemple.

Si vous avez la curiosité d'examiner le contenu de la définition de SEIZE, vous n'y retrouverez que la valeur 16 précédée d'une adresse située au CFA.

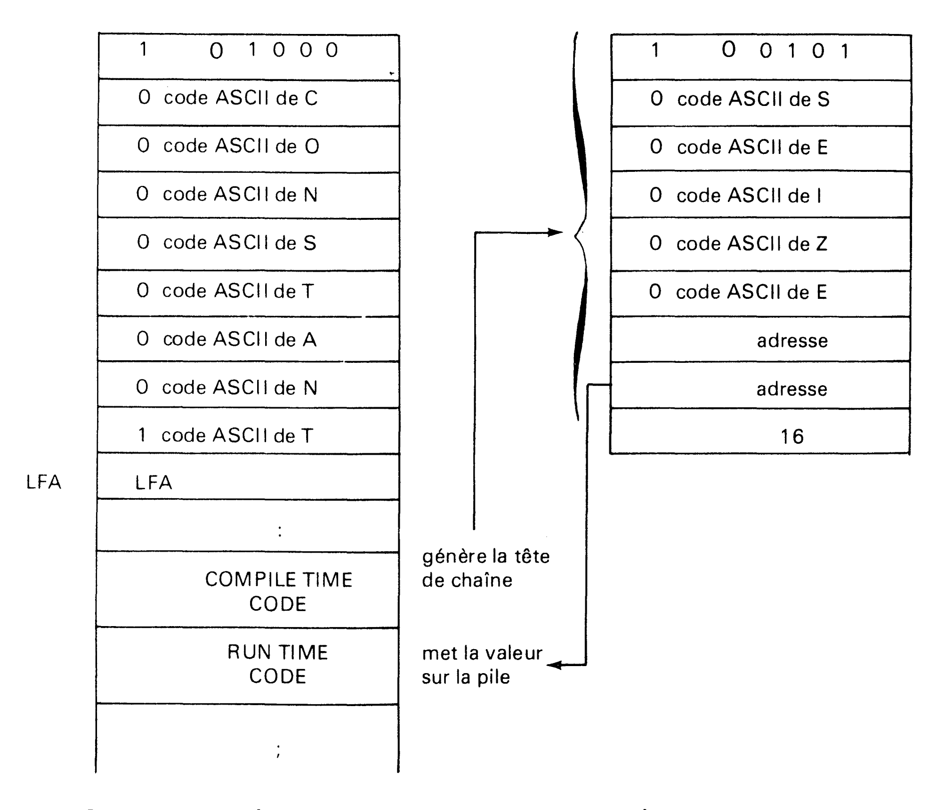


A la lumière de la présentation qui a été faite de l'interpréteur, vous savez que lorsque vous tapez :

SEIZE [Return] OK

l'interpréteur va exécuter le code pointé par le CFA. C'est donc cette exécution qui va placer la valeur de la constante sur la pile.

Ce code pointé par le CFA des constantes n'est autre que la deuxième partie de la définition du mot CONSTANT, la première étant celle qui génère la tête de chaîne.



Les mots de définition que nous allons vous présenter comportent deux parties :

— La première, exécutée lorsqu'on les utilise pour créer un mot, crée une tête de chaîne.

— La seconde est appelée lorsqu'on exécute les mots ainsi créés. Elle effectue la totalité du traitement donné.

L'intérêt d'un tel outil est par exemple le pouvoir structurer des données : le traitement choisi dépendra du type des données.

Pour les constantes, le traitement met la valeur de la constante sur la pile. Pour les variables, il met l'adresse de la valeur sur la pile (PFA).

Rappelons que pour les constantes comme pour les variables, leur valeur est au PFA de la définition.

On connaît l'adresse de la valeur des variables : on pourra donc facilement les modifier grâce aux mot @ et ! . Pour les constantes, nous n'avons accès qu'à la valeur, c'est ce qui rend leur modification plus délicate.

Les USER sont assez semblables aux variables par le fait qu'elles retournent l'adresse de leur valeur sur la pile, mais leur implantation mémoire est différente. En effet, les valeurs sont localisées dans une table à un emplacement fixe dans la mémoire. L'adresse de la valeur dans la table se trouve au PFA de la définition de la USER.

L'intérêt de cette table fixe réside dans l'implantation du multitasking : les USER VARIABLES sont en effet les seuls mots du FORTH de base susceptibles d'êtres différents d'un utilisateur à l'autre.

VI.2.3. Outils de définition des mots de définition

Il s'agit de <BUILDS et DOES>, toujours employés par paire. La structure générale de la définition classique d'un mot de définition est sur le modèle suivant :

```
: MOTDEDEFINITION
                    <BUILDS
                              Traitement 1
                    DOES>
                              Traitement 2
; [Return] OK
```

C'est <BUILDS qui créera effectivement la tête de chaîne. A l'exécution du mot de définition, la tête de chaîne créée portera le nom du mot qui le suit.

Exemple :

Supposons que nous avons défini le mot de définition DEFINE. Après avoir utilisé la syntaxe suivante :

DEFINE OBJET [Return] OK

On trouvera dans le dictionnaire une nouvelle définition portant le nom OBJET.

L'exécution de "Traitement 1" a lieu lors de l'exécution du mot de définition, après la création de la tête de chaîne.

Nous comprendrons mieux sa fonction dans le paragraphe suivant.

La finalité des mots de définition est d'associer un traitement donné à l'exécution des mots définis par leur intermédiaire.

Le traitement en question est représenté ici par "Traitement 2". Il n'est donc pas à exécuter lors de l'exécution du mot de définition (*exemple*: DEFINE OBJET). Il faut bien comprendre que ce traitement est résident dans la définition du mot de définition et non pas dans celle du mot défini, où il ne figure que par un pointeur, placé là lors de l'exécution de DOES>.

La fonction de DOES> est de dérouter l'interpréteur sur "Traitement 2" lors de l'exécution du mot défini après avoir placé sur la pile le PFA du dit mot.

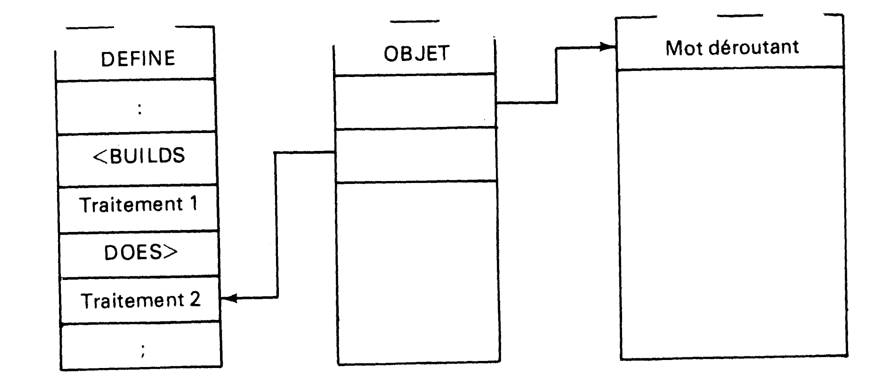
Ces renseignements suffisent pour la création et l'utilisation de mots de définition. Si vous êtes curieux de connaître le fond du problème, nous allons maintenant aborder le fonctionnement interne de DOES>.

Exécutons pas à pas le mot défini OBJET. Nous savons que l'interpréteur va considérer le contenu de la définition d'OBJET comme une suite de CFA de mots à exécuter. Dans notre cas, ce mode de fonctionnement n'est pas adapté si l'on pense par exemple aux variables qui contiennent une valeur numérique qui n'est pas un CFA. Le mot dont le CFA est situé au CFA d'OBJET devra donc dérouter l'exécution vers "Traitement 2" et non vers la suite de la définition d'OBJET, en mettant au passage le PFA+2 d'OBJET sur la pile afin que "Traitement 2" sache d'où vient l'appel.

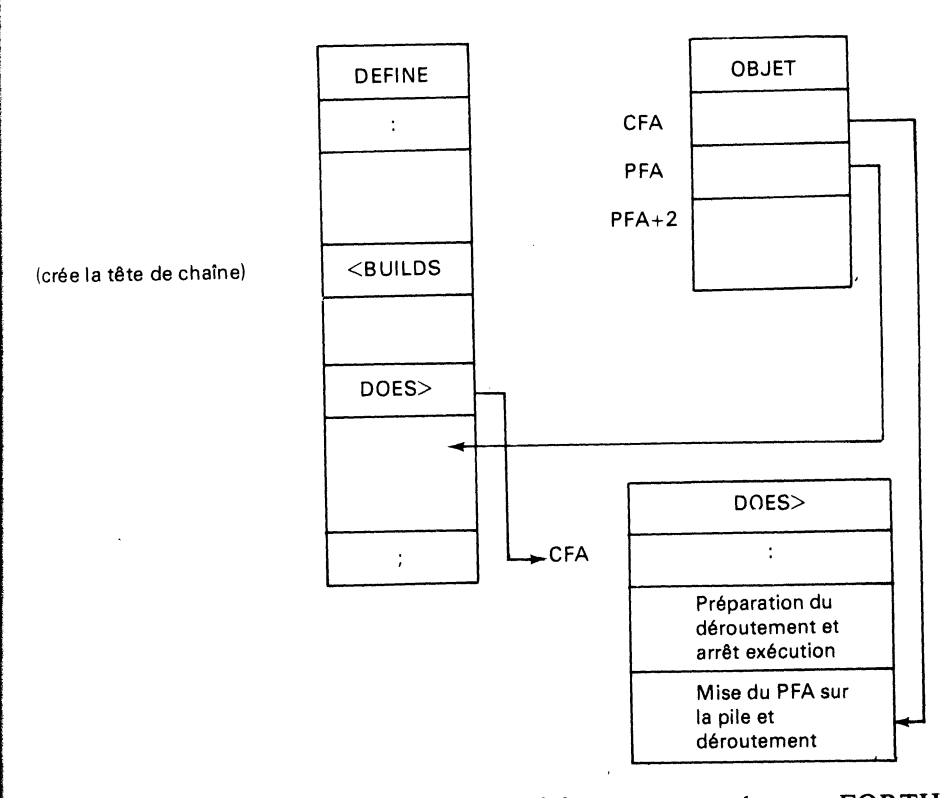
Le début de la définition d'OBJET comprend donc deux adresses: le CFA du mot "déroutant" et l'adresse du déroutement, c'est-à-dire le début de "Traitement 2".

<BUILDS réserve deux emplacements après la tête de chaîne. C'est l'exécution de DOES> qui placera les deux adresses "pointeur" dans ces deux cases mémoire.

Tout est donc maintenant assemblé selon le schéma suivant:



En fait, les choses sont encore plus subtilement organisées. Le mot "déroutant" est inclus dans la définition de DOES> et le schéma correct est le suivant:



Nous vous recommandons de vous référer au source de votre FORTH pour plus ample information.

A titre d'application, voyons comment nous pourrions écrire les mots de définition CONSTANT et VARIABLE en FORTH.

Il est clair que la clé est de trouver "Traitement 1" et "Traitement 2".

CONSTANT

Traitement 1	Traitement 2
Placer la valeur numérique présente sur la pile au premier emplacement libre dans le dictionnaire (PFA+2)	PFA+2 étant sur la pile, chercher son contenu et le placer sur la pile.
C'est-à-dire : ,	C'est-à-dire : @

```
: CONSTANT
          <BUILDS
                 ,
          DOES>
                @
; [Return] OK
```

VARIABLE

Traitement 1	Traitement 2
Même chose que pour CONSTANT.	L'adresse du début de la valeur est directement sur la pile, il n'y a donc aucun traitement supplémentaire.

C'est-à-dire : ,

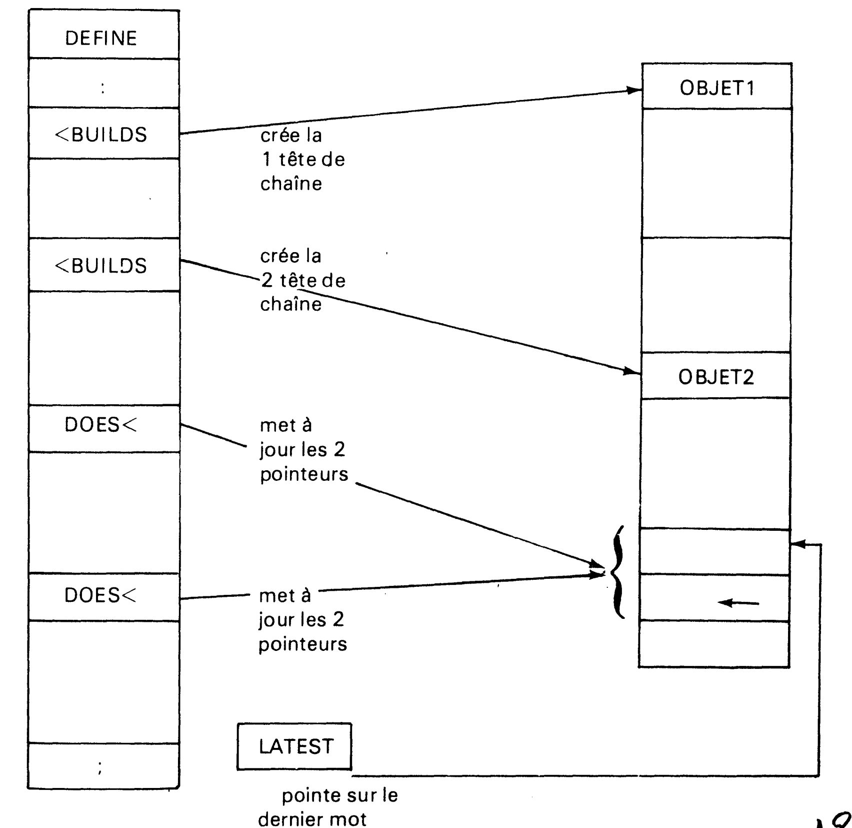
```
: VARIABLE
          <BUILDS
                 ,
          DOES>
; [Return] OK
```


Il est possible dans la définition d'un mot de définition d'intercaler un traitement avant <BUILDS. Il ne sera bien sûr exécuté que lors de l'exécution du mot de définition.

Exemple :

```
: DEFINE
        ." JE DEFINIS"
        <BUILDS
                "Traitement 1"
        DOES>
                "Traitement 2"
; [Return] OK
```

Par contre, il n'est pas possible d'imbriquer des structures <BUILDS et DOES>. En effet, DOES> invoque dans son exécution le mot LATEST qui retourne le CFA du dernier mot du vocabulaire courant, donc du mot en cours de définition. Nous aboutirions donc au problème suivant :



… deux… DO… ne t…aille p… …r le … grou… …e p…urs.

VI.2.4. Exemples d'utilisation : STRING, ARRAY

L'outil indispensable pour créer des structures de données complexes est ALLOT qui réserve des emplacements mémoire à partir de la position courante du pointeur de dictionnaire (DP). Il prend sur la pile le nombre d'octets à réserver.

```
: ALLOT DP@  + DP! ; [Return] OK
```

A noter que DP@ n'est autre que le mot HERE.

ALLOT n'est pas lié à priori à l'action de définition.

STRINGS

Nous nous proposons maintenant d'implanter en FORTH la structure chaîne de caractères (STRING).

Analyse :

Cette structure présentera des caractéristiques classiques (cf Basic par exemple).

— La capacité maximum de la chaîne est donnée au départ.

— On peut y accéder soit globalement, soit grâce à un système de sous-chaînes.

Pour cela, nous implanterons la valeur dans la définition, avec deux index donnant la taille maximum de la chaîne et le nombre de caractères déjà affectés.

	chaîne
CFA	
PFA	
PFA+2	Dimension
PFA+4	Utilisés
PFA+6	valeurs

Notre définition doit nous permettre d'utiliser la syntaxe suivante :

Pour la définition de la chaîne :

```
Nb carac.   STRING  Nom de la chaîne
```

Pour l'accès, l'exécution du nom de la chaîne retournera sur la pile :

```
(------Adr.chaîne Dimension Utilisés)
```

Implantation

Le traitement associé à <BUILDS sera :

— Prendre Dimension sur la pile, la ranger dans la définition.

— Initialiser à zéro le nombre de caractères utilisés.

— Réserver le nombre de caractères de la dimension.

Le traitement associé à DOES> consiste à calculer et empiler les valeurs demandées à partir de PFA+2 présent sur la pile.

```
: STRING                    (dim----)
     <BUILDS
             DUP ,          (dim---)  range la dimension
             0 ,            (dim---)  range le Nb de caractères
                                      utilisés
             ALLOT          (---)     réserve la dimension
     DOES>
          4 + DUP 4 -   (Ad.chaîne Ad.dim.---)
          DUP @ SWAP    (Ad.chaîne dim. Ad.dim.---)
          2 + @         (Ad.chaîne dim. utilisés--)
; [Return] OK
```

Pour affecter globalement une valeur à cette chaîne, nous avons choisi une méthode assez générale : la chaîne devra être délimitée par des guillemets afin d'autoriser les caractères blancs en début, milieu et fin de chaîne.

Il est utile de disposer d'un mot qui recherche en mémoire un caractère donné avec une adresse de départ, et une adresse limite d'exploration.

On lui place dans la pile le code ASCII du caractère cherché, l'adresse de début et le nombre de caractères à scruter. Il retourne après exécution l'adresse du caractère cherché en cas de succès et un flag de succès ou d'échec.

```
: POS                       ( C  Ad.  Max---Ad  Flag )
    0 SWAP                  ( C  Ad.  0  Max.--- )
    0 DO
        DROP                ( C  Ad.--- )
        OVER  OVER  @ -0=   ( C  Ad.  flag--- )
        IF
          1  LEAVE          ( C  Ad. 1--- )
        ELSE
            1 +  0          ( C  Ad. + 1  0--- )
        THEN
    LOOP
    ROT  DROP               ( Ad.  flag--- )
; Return OK
```

A partir de POS, construisons un mot qui teste la validité d'une constante alphanumérique entre guillemets, en prenant sur la pile une adresse de début, la longueur maximum tolérée de la constante, et qui retourne sur la pile cette même adresse, la longueur effective de la chaîne et un flag (succès-échec).


```
: ?STRING               ( Ad  Max-  Ad  long  flag )
    OVER C@             ( Ad  Max  C1--- )
                        ( prend le 1° caractère )
    34  -               ( Ad  Max  flag--- )
                        ( le compare à " )
    IF
      DROP  0  0        ( Ad  0  0--- )
                        ( ne commence pas par " )
     ."  MAUVAIS SEPARATEUR"
    ELSE
        OVER  34  ROT
                  ROT1 ( Ad"  Max  Ad--- )
        SWAP  POS       ( Ad  Ad'  flag---)
                        ( Cherche le second " )
    IF                  ( Ad  Ad'---)
      OVER - 1 - 1      ( Ad  long  1--- )
                        ( Succès )
    ELSE
        DROP  0  0      ( Ad  0  )--- )
        ."  CHAINE TROP LONGUE"
    THEN
  THEN
; Return OK
```

On peut maintenant facilement construire $=

```
: $=                    ( Ad  Max  Util--- )
   DROP                 ( Ad Max ---)
   TIB  IN  +           ( Ad  MAX  Ad.début--- )
   SWAP                 ( Ad  Ad.début  MAX--- )
   ?STRING              ( Ad  Ad.début  Long  Flag--- )
   IF                   ( Ad Ad.début Long--- )
     1  0  D+  ROT  2 DUP  2-  ! SWAP ( Mise à jour de Util )
    THEN
    CMOVE  QUIT
; Return OK
```

... .rois ...es d... ...isati... .e $= ... justif... par l... yse ...n-dante suivante:

— Transférer une chaîne du Terminal Input Buffer dans la variable désignée, donc:

— Être capable de tester la validité du mot suivant comme constante alphanumérique: présence de séparateurs, longueur⩽dimension maximale et donc:

— Rechercher un caractère donné à partir d'une adresse mémoire avec une limite d'exploration.

Exemple:

```
50 STRING TEST [Return] OK
TEST $= "CHAINE DE CARACTERES" [Return] OK
```

A partir de maintenant, il est extrêmement simple de réaliser l'affichage d'une chaîne à l'écran:

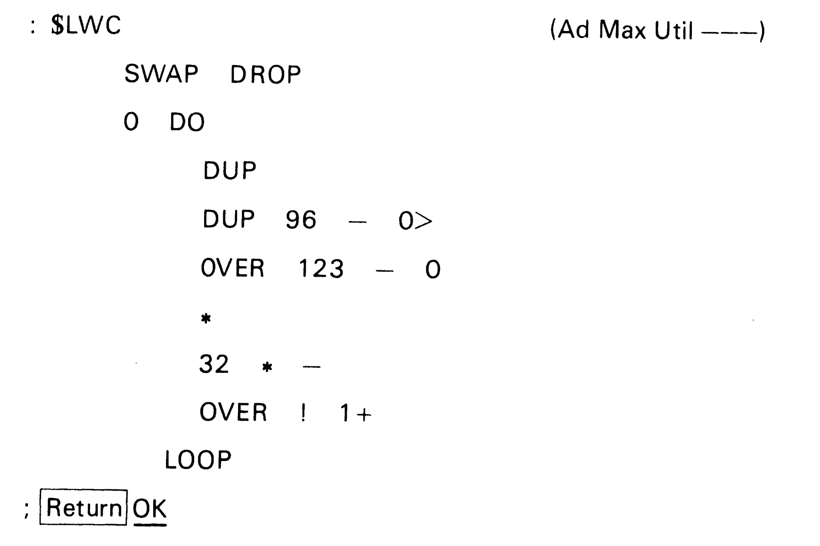
```
: $DISP          (Ad Max Util–––)
      SWAP DROP TYPE
; [Return] OK
```

De même, les fonctions classiques de traitement des chaînes s'implantent de façon suivante:

Passage minuscules-majuscules.

```
: $UPC                        (Ad Max Util–––)
     SWAP DROP                (Ad Util–––)
     0   DO                   (Ad –––)
          DUP @               (Ad C –––)
          DUP  64  –  0>      (test: C est-il un)
          OVER  91  –  0<     (caractère alphanumérique ?)
          *                   (Ad C flag –––)
          32  *  +            (Ad C+(flag*32) –––)
          OVER  !  1+         (Ad+1 –––)
     LOOP
; [Return] OK
```


Passage majuscule-minuscule.



```
: $LWC                        (Ad Max Util –––)
     SWAP  DROP
     0   DO
          DUP
          DUP  96  –  0>
          OVER  123  –  0
          *
          32  *  –
          OVER  !  1+
     LOOP
; [Return] OK
```

ARRAYS

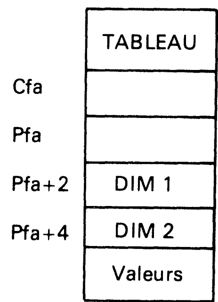
Autre exemple d'application de <BUILDS et DOES>: les matrices à deux dimensions.

Analyse:

— A la création, les deux dimensions sont données.

— On peut y accéder soit globalement, soit grâce à un système de sous-matrices.

Pour cela, on implantera le tableau dans la définition, en le faisant précéder par deux valeurs correspondant aux deux dimensions.



Notre définition devra nous permettre d'utiliser les syntaxes suivantes:

— Pour la création:

DIM1 DIM2 MATDEF Nom du tableau

— Pour l'accès, l'exécution de

ind1 ind2 Nom du tableau

retournera sur la pile l'adresse de l'élément désigné.

Implantation :

Le traitement associé à <BUILDS sera :

— prendre les deux dimensions sur la pile, les ranger dans la définition,

— réserver dim1 × dim2 cellules dans la définition, afin d'y ranger ultérieurement la matrice.

Le traitement associé à DOES> consiste à tester la validité des indices ind1 et ind2 par rapport à dim1 et dim2, puis calculer l'adresse de l'élément demandé par une opération arithmétique du type :

PFA+6 + (ind1 × Dim1) + ind2

```
: MATDEF                         ( Dim1 Dim2--- )
      <BUILDS
          OVER
          OVER
          *   ALLOT
      DOES>                      ( Ind1 Ind2 PFA+2 --- )
          >R                     ( Ind1 Ind2 --- )
          2 DUP                  ( Ind1 Ind2 Ind1 Ind2 --- )
          I                      ( Ind1 Ind2 Ind1 Ind2 Dim1 --- )
          I  2+                  ( Ind1 Ind2 Ind1 Ind2 Dim1 Dim2 --- )
          ROT  -   0>
          ROT
         ROT  -   0<             ( Ind1 Ind2 Flag1 Flag2 --- )
          *
          IF
                   I  ROT  *  +
                   R> 4   +               ( Offset PFA+6 --- )
                   +                      ( Adresse --- )
          ELSE
                   R> DROP  2DROP
                   ." MAUVAIS INDICES"
          THEN
; Return  OK
```


A partir de maintenant, chaque élément de cette matrice à deux dimensions se comporte individuellement comme une variable ordinaire, ce qui rend lecture et écriture des éléments aussi aisée que dans d'autres langages.

Pour pousser plus avant l'exploitation de cette nouvelle structure de données, on peut créer le mot simple suivant :

```
: MATDISP
        [COMPILE]                    ( PFA--- )
        6  +  DUP  >R                ( CFA+8 --- )
        I  4  -  @                   ( CFA+8 Dim1 --- )
        R>  2  -  @                  ( CFA+8 Dim1 Dim2 --- )
        0  DO
            SWAP  OVER               ( Dim1 Ad Dim1 --- )
            0  DO
                   DUP @ .
                   2+
            LOOP
            SWAP                     ( Ad' Dim1 --- )
        LOOP
        2DROP
; Return OK
```

VI.3. EXÉCUTION PAR INDIRECTION

Continuons dans le cadre de l'exemple précédent.

Dans nombre de fonctions élémentaires du calcul matriciel, on retrouve toujours le même algorithme de balayage des lignes et des colonnes. Seul le traitement situé au cœur des deux boucles est susceptible de changer.

ΛT FI

```
    PREPARE                     (PFA+6 dim1 dim2 ---)
    0 DO                        (PFA+6 dim1 ---)
        SWAP OVER               (dim1 PFA+6 dim1 ---)
        0 DO                    (dim1 PFA+6 ---)
            DUP  TRAITEMENT  2  +
        LOOP
        SWAP                    (PFA+6 dim1 ---  )
    LOOP
    2DROP
; Return OK
```

avec :

```
: PREPARE
    [COMPILE] '          (PFA ---)
    6  +  DUP  >R        (PFA+6 ---)
    I  4  -  @           (PFA+6 dim1 ---)
    R>  2  -  @          (PFA+6 dim1 dim2 ---)
; Return OK
```

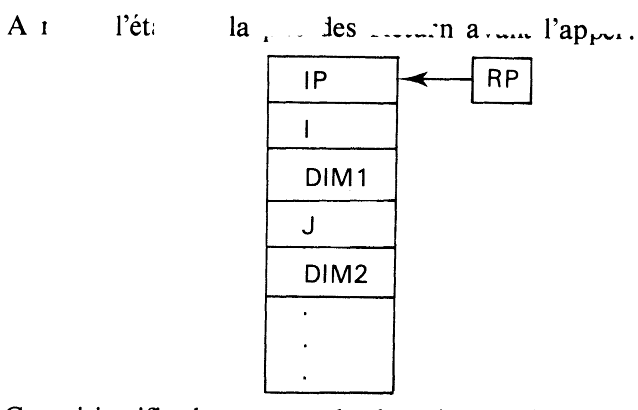
Le traitement pour la mise à une valeur constante s'écrira :

```
: INIT
    CONST SWAP !
; Return OK
```

De même la matrice identité se générera automatiquement en prenant comme traitement :

```
: IDENT
    R> I J ROT >R
    = SWAP !
; Return OK
```


A ı l'ét: la , des ...n a... l'app...



Ce qui justifie de sauvegarder le pointeur d'interprétation dans la pile de données avant de pouvoir accéder à I et J.

Les opérations scalaires, tels que addition de la constante CONST à tous les éléments, s'écrivent :

```
: PLUS
    CONST OVER @ + SWAP !
; Return OK
```

L'idée de l'exécution par indirection est de pouvoir paramétrer un traitement dans un algorithme par l'intermédiaire d'une variable.

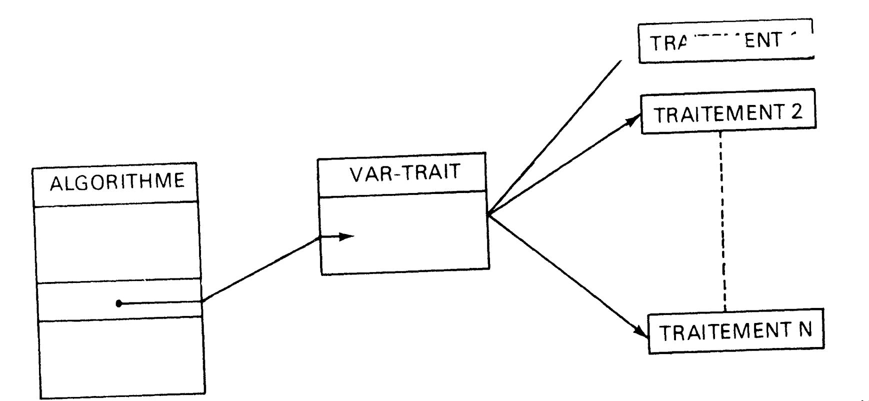
L'outil de base pour réaliser ce type de miracles est le mot EXECUTE qui exécute spontanément le mot dont le CFA est sur la pile.

Ce mot a déjà été cité lors de la présentation du mécanisme interne de l'interpréteur.

La mise en œuvre d'une exécution par indirection suppose deux phases :

— Création et préparation d'une variable contenant le CFA du mot correspondant au traitement choisi.

— Dans l'algorithme, remplacement du traitement par VAR-TRAIT EXECUTE.



Il s'agit bien d'une exécution par indirection puisqu'il est possible d'agir extérieurement sur la fonction de l'algorithme.

```
0 VARIABLE VAR-TRAIT

: MAT FN
        PREPARE
        0 DO
            SWAP OVER
            0 DO
                DUP VAR-TRAIT    EXECUTE 2+
            LOOP
            SWAP
        LOOP
        2DROP
; Return OK
```

Les fonctions :

```
: T-DISP @ . ; Return OK
: T-INIT CONST SWAP ! ; Return OK
: T-PLUS CONST OVER @ + SWAP ! ; Return OK
```

Les traitements correspondants :

```
: MATDISP ' T-DISP VAR-TRAIT ! MATFN ; Return OK
: MATINIT ' T-INIT VAR-TRAIT ! MATFN ; Return OK
: MATPLUS ' T-PLUS VAR-TRAIT ! MATFN ; Return OK
```

En FORTH, il est toujours possible de travailler à deux niveaux :

— Soit dans la lignée des langages classiques, auquel cas le fonctionnement de l'interpréteur est transparent au programmeur.

— Soit à un niveau plus performant, en intégrant l'application dans la philosophie du langage.

Mot	*Syntaxe*	*Définition*
IMMEDIATE	IMMEDIATE	Rend le dernier mot créé dans le dictionnaire immédiat, c'est-à-dire que ce mot sera exécuté même lors de la compilation d'une définition où il apparaît.
BRANCH	BRANCH "adresse relative"	Effectue un branchement à l'adresse relative (par rapport à la position du BRANCH).
0BRANCH	0BRANCH "adresse relative"	Effectue un branchement conditionnel (flag nul sur la pile) à l'adresse relative (par rapport à la position du 0BRANCH).
COMPILE	COMPILE "mot"	A l'exécution du mot dont la définition contient COMPILE, le CFA de "mot" sera chargé à la suite du dictionnaire.
[	["suite de mots"]	L'interpréteur passe en mode exécution. Ce mot est toujours utilisé en paire avec le mot].
]	["suite de mots"]	L'interpréteur repasse en mode compilation. Ce mot est toujours utilisé après le mot [.
[COMPILE]	[COMPILE] "mot"	Lors de la compilation, le mot immédiat "mot" n'est pas exécuté mais compilé. L'interpréteur ne tient pas compte de la valeur de son Precedence bit.
C,	"octet" C,	La valeur sur un octet ("octet") qui précède C, est chargée dans la première cellule libre du dictionnaire.
,	"valeur" ,	La valeur sur deux octets ("valeur") qui précède le mot, est chargée dans la première cellule libre du dictionnaire.
<BUILDS	<BUILDS "Traitement1" DOES> "Traitement2"	C'est l'entête du traitement correspondant au COMPILE-TIME. Il est toujours utilisé en paire avec DOES>.
DOES>	<BUILDS "Traitement1" DOES> "Traitement2"	Indique la fin du traitement pour le COMPILE-TIME et représente l'entête du traitement ("Traitement2") correspondant au RUN-TIME. Il est toujours utilisé en paire avec <BUILD. Après "Traitement2" aucun traitement ne peut être intercalé avant le mot de fin de définition ; .
ALLOT	"nombre" ALLOT	Ce mot réserve à la suite du dictionnaire le nombre "nombre" d'octets.

EXERCICE 1 :

Définissez en FORTH les mots , et C, à l'aide du mot ALLOT.

EXERCICE 2 :

Définissez en FORTH le mot COMPILE.

Indication : le pointeur d'interprétation, au sommet de la pile des return, pointe sur le mot suivant COMPILE dans la définition en cours de compilation.

EXERCICE 3 :

Définissez en FORTH les mots qui constituent la structure de contrôle IF ––– ELSE ––– THEN (ou ENDIF).

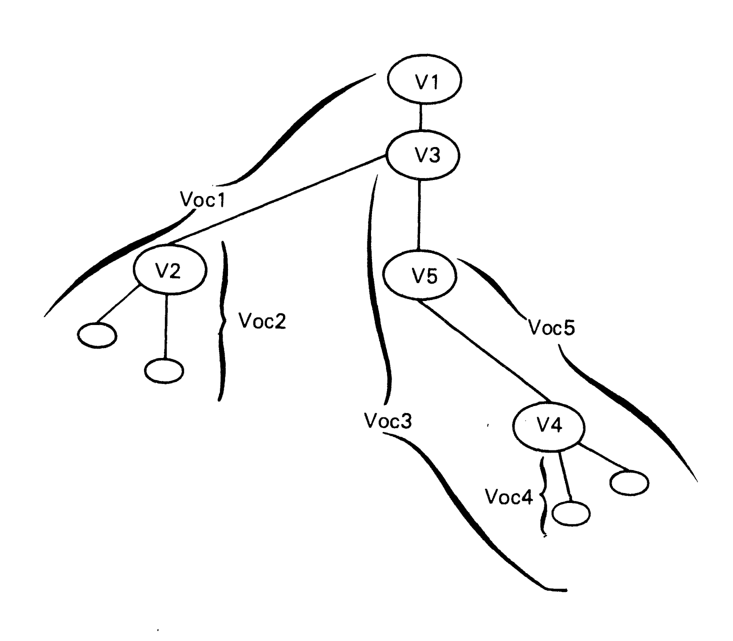
VII

LES PROPRIÉTÉS PARTICULIÈRES

VII.1. LES VOCABULAIRES

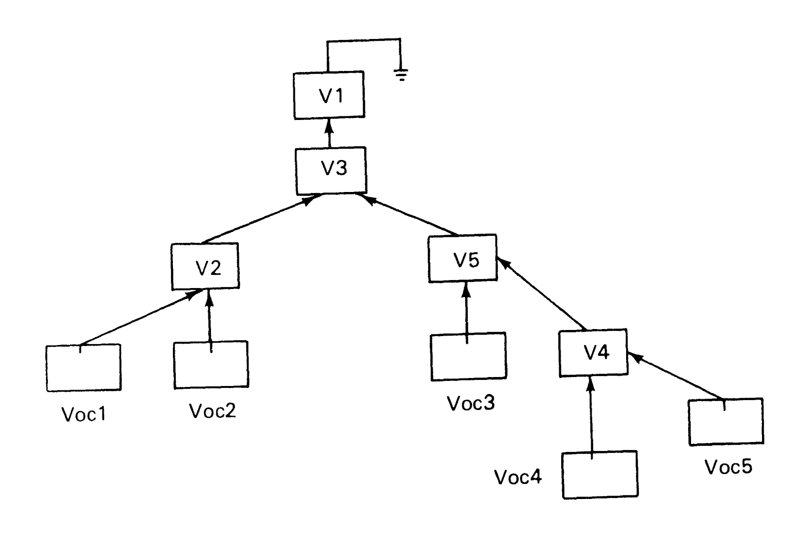
Comme nous l'avons déjà vu à plusieurs reprises, le dictionnaire est ramifié en vocabulaires.

Chaque vocabulaire présente une structure de chaîne linéaire partant du nom du vocabulaire et allant continûment jusqu'à une feuille terminale.



La recherche d'un mot dans un vocabulaire se fait toujours depuis sa feuille terminale en remontant vers la racine. Elle ne s'arrête pas à la rencontre du nom du vocabulaire, mais enchaîne bel et bien jusqu'à la racine de l'arbre. C'est en ces termes que l'on entend que tout vocabulaire contient le noyau FORTH.

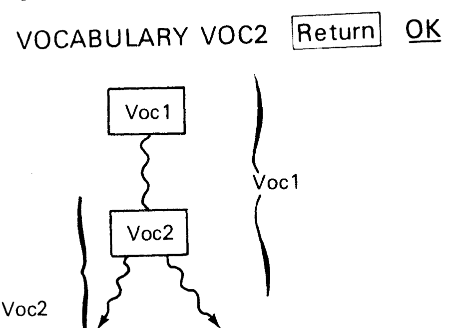
Cette organisation est tout à fait cohérente avec la structure des définitions dans le dictionnaire, puisqu'un seul pointeur arrière suffit à l'implanter :



Pour créer un vocabulaire, on utilise le mot de définition VOCABULARY qui, dans le vocabulaire courant, construit une tête de chaîne au nom du nouveau vocabulaire.

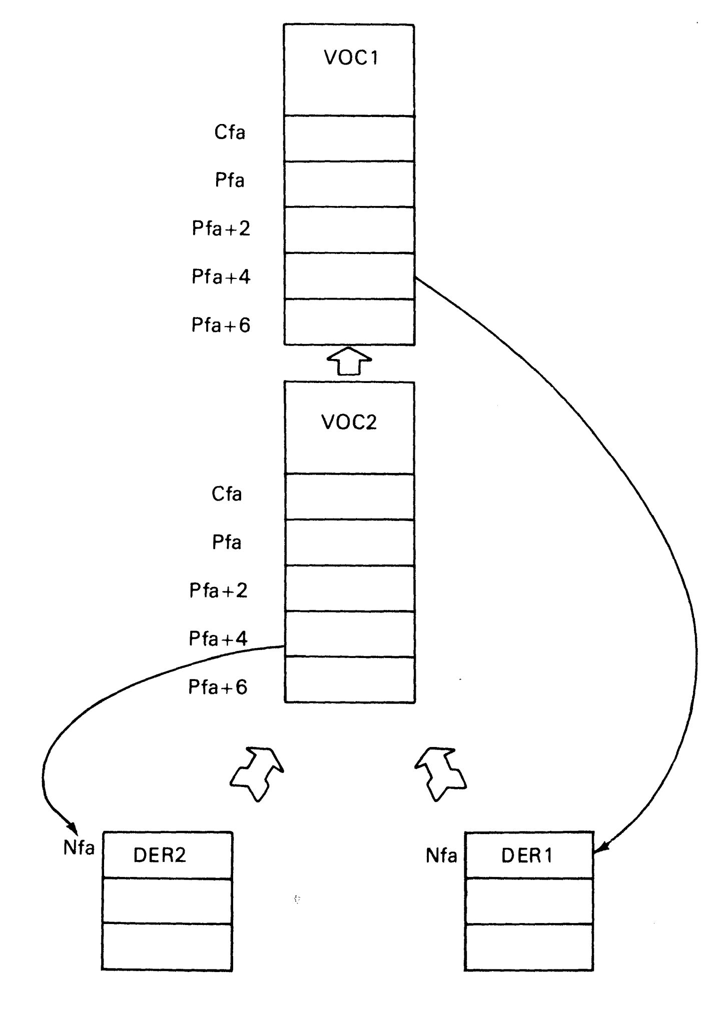
Exemple :

VOCABULARY VOC2 [Return] OK



Le nombre de vocabulaires est illimité.

A chaque vocabulaire correspond une tête de chaîne dont l'adresse est simplement rangée dans la définition du nom du vocabulaire :



Ces vocabulaires peuvent être utilisés de deux manières : en tant que vocabulaire courant ou de définition, et c'est dans ce vocabulaire choisi que s'ajouteront les nouvelles définitions, ou comme vocabulaire de contexte, et c'est dans ce vocabulaire choisi que l'interpréteur effectuera les recherches des mots qui lui sont proposés.

Pour mettre en pratique cette notion, on dispose de deux user variables, CURRENT et CONTEXT.

CURRENT pointe en permanence sur la zone de la définition du vocabulaire courant qui contient le NFA du dernier mot créé dans ce vocabulaire. Cette zone est située à l'adresse PFA+4 de la définition.

Pour trouver ce NFA, il faut donc faire une indirection de type CURRENT @ @ : c'est le mot LATEST.

```
: LATEST CURRENT @ @ ; [Return] OK
```

A chaque nouvelle définition, CURRENT reste donc inchangé.

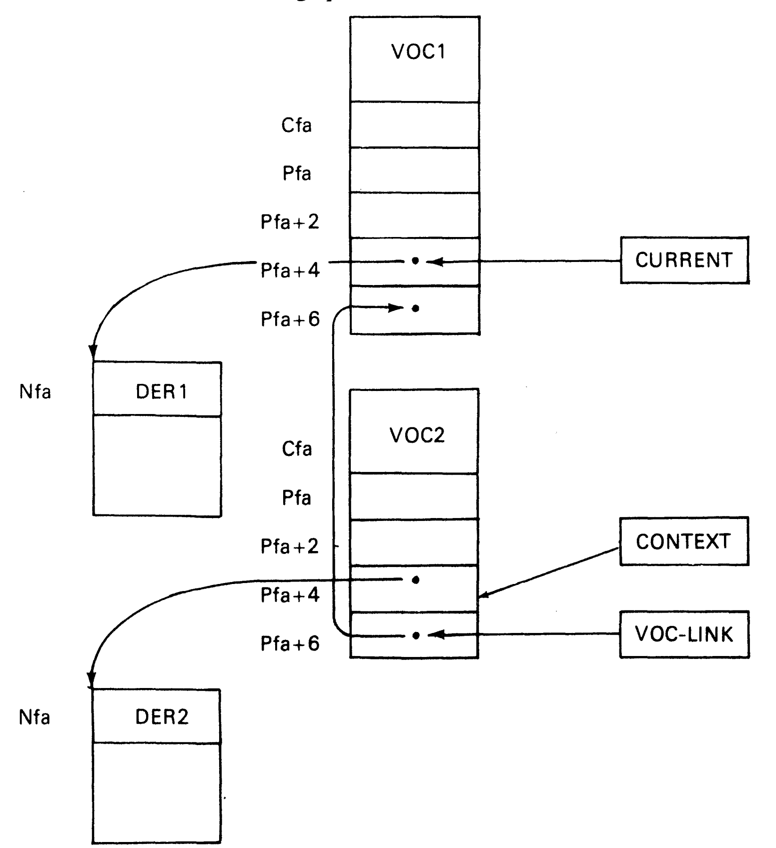
La mise à jour du pointeur vers le dernier mot créé dans le vocabulaire courant se fait par une opération du type CURRENT @ ! .

CONTEXT pointe sur une zone analogue, cette fois dans la définition du nom du vocabulaire de contexte.

Une autre facilité de FORTH permet de connaître l'ensemble des vocabulaires résidents à un instant donné. Pour cela, les noms des différents vocabulaires sont chaînés entre eux de façon chronologique.

La tête de cette chaîne est contenue dans la user variable VOC-LINK, qui pointe sur une zone située à l'adresse PFA+6 dans la définition du nom du dernier vocabulaire créé.

Cette zone pointe elle-même sur la zone analogue dans la définition du nom du vocabulaire chronologiquement antérieur.

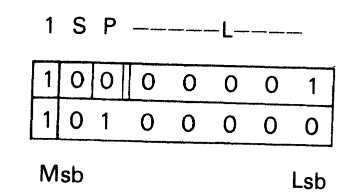


Bien que tous les points n'aient [illegible] s en [illegible] été [illegible]cis, [illegible] pou[illegible] tout de même vous donner une définition de VOCABULARY pour vous fixer les idées. Nous reprendrons ensuite cellule par cellule toute la tête de chaîne générée.

```
: VOCABULARY
     <BUILDS                  ( Création de la tête de chaîne,
                                remplissage de Cfa et de Pfa )
        A081 ,                ( Initialisation de Pfa+2 )
        CURRENT @ CFA ,       ( Initialisation de Pfa+4 )
        HERE VOC-LINK @ ,     (Initialisation de Pfa+6, et lien vers
                                vers le dernier vocabulaire créé )
        VOC-LINK !            ( Mise à jour de VOC LINK )
     DOES>                    ( Placera le Pfa sur la pile lors
                                de l'exécution du mot )
        2+ CONTEXT !          ( fera pointer CONTEXT sur le
                                vocabulaire choisi )
; [Return] OK
```

A la vue de cette définition, la première question que vous êtes en droit de vous poser est la fonction de 81A0 placé à l'adresse Pfa+2 de la définition générée.

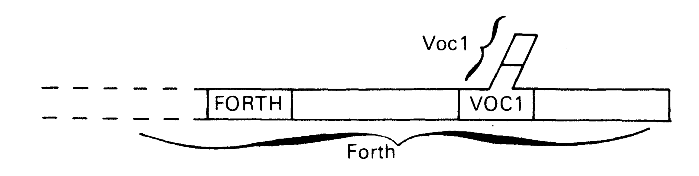
Les rares ouvrages qui se hasardent à élucider 81A0 se contentent de l'appeler "Dummy header" (tête de chaîne fictive). En effet, 81A0 correspond à :



On peut donc comprendre ces deux octets comme la tête de chaîne d'un mot dont le nom est un caractère blanc (Space).

C'est effectivement comme cela que l'interpréteur le comprend, et il nous reste à expliquer sa fonction.

Supposons que notre FORTH ait l'aspect suivant :



A un moment, vous avez créé VOC1, et créé un certain nombre de définitions sous ce vocabulaire. Ensuite, de nouveau sous FORTH, vous avez créé d'autres mots. Placez vous maintenant dans VOC1, et faites VLIST sur votre terminal. Vous verrez les définitions apparaître dans l'ordre suivant :

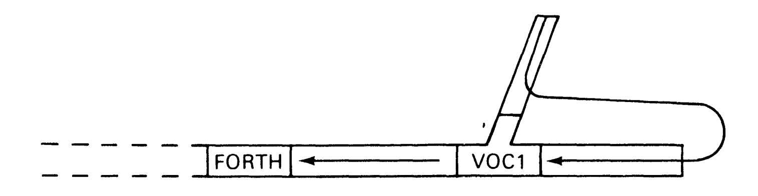
Définitions faites sous VOC1

(trois espaces)

Définitions faites sous FORTH

VOC1

Reste des définitions



L'arborescence a donc été parcourue d'une façon inattendue.

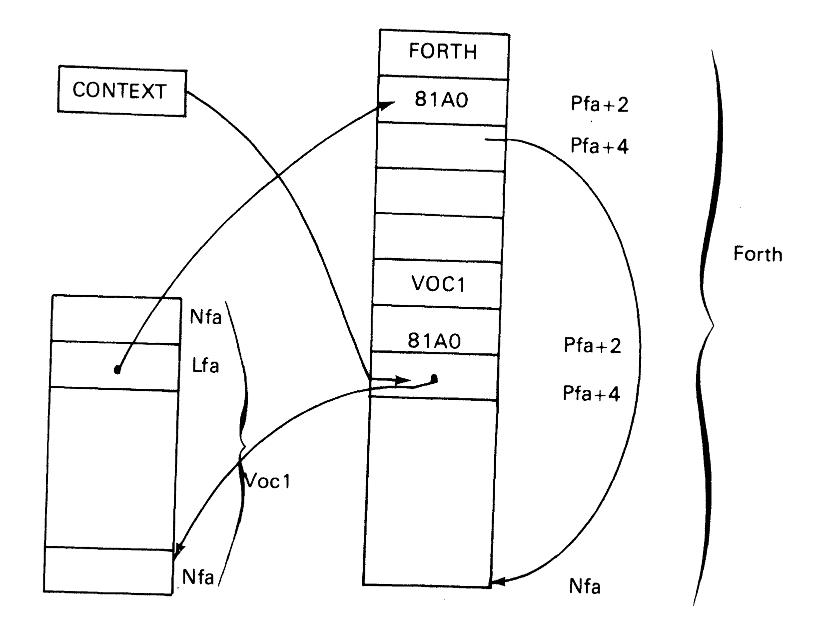
Sachant que VLIST ne fait qu'un enchaînement systématique des Lfa jusqu'à en rencontrer un qui soit nul, on peut envisager deux explications possibles :

— Le Lfa du premier mot créé dans VOC1 pointe sur le Nfa du dernier mot créé sous FORTH. Cela suppose que chaque nouveau mot créé sous FORTH nécessite une mise à jour de ce Lfa.

— Ou bien, sachant que dans la définition du vocabulaire FORTH, le contenu de la cellule d'adresse Pfa+4 pointe en permanence sur le Nfa du dernier mot créé dans ce vocabulaire, il suffirait de prendre cette cellule pour un Lfa, et de faire pointer sur le Nfa le mot fictif correspondant le Lfa

…prem… mot … OC1 cha… ser… …ors … réal… …a tê… chaîne fictive en question n'est autre que 81A0.

On peut maintenant faire un schéma plus complet :



Toute recherche dans le dictionnaire commence à l'adresse obtenue en faisant CONTEXT @ .

On est donc sur le mot fictif du vocabulaire de contexte. Son LFA pointe sur le NFA du dernier mot du vocabulaire.

Pour le premier mot créé dans VOC1, son LFA pointe sur le NFA du mot fictif dans le vocabulaire sous lequel VOC1 a été créé.

On trouve bien le passage par ces têtes de chaîne fictives dans VLIST, qui apparaissent sous forme d'un caractère blanc.

Au passage, voici l'algorithme de parcours de VLIST, qui est d'ailleurs identique pour toute recherche dans le dictionnaire :

```
      CONTEXT @ @         ( Nfa du dernier mot créé dans le
                            vocabulaire de contexte. )
      BEGIN
           DUP            ( NFA NFA ––– )
           TRAITEMENT     ( Doit consumer un Nfa )
           PFA            ( PFA ––– )
           LFA            ( LFA ––– )
            @             ( (LFA) ––– )
           DUP            ( (LFA) (LFA) ––– )
           0=             ( (LFA) flag ––– )
      UNTIL
      DROP
; Return OK
```

L'application directe est VLIST où le traitement à effectuer consiste à faire un éventuel passage à la ligne, et à éditer le nom du mot :

```
: TRAITEMENT
      OUT@                ( Donne le nombre de caractères émis )
      C/L                 ( Donne le nombre de caractères/ligne )
      >
      IF
           CR             ( Passage à la ligne )
           0 OUT !        ( Reset du compteur de caractères )
      ENDIF
      DUP
      ID.                 ( Affiche le nom à partir du NFA )
      SPACE
; Return OK
```


R[illegible] de la [illegible]nition [illegible] VOCABULARY.

La séquence :

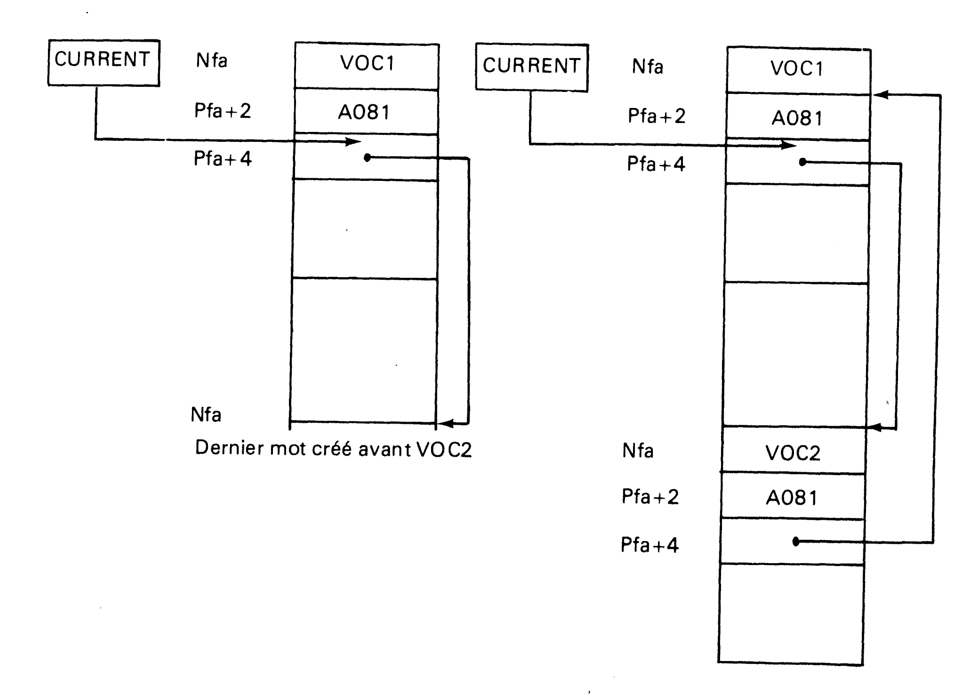
```
A081 ,
CURRENT @ CFA ,
```

peut se décomposer en deux temps :

— Création de la tête de chaîne fictive.

— Initialisation du LFA de cette tête de chaîne fictive : on le fait pointer sur le NFA du mot fictif dans le vocabulaire courant, sous lequel ce nouveau vocabulaire est crée. L'action du mot CFA n'est autre que 2–, comme nous l'avons déjà vu.

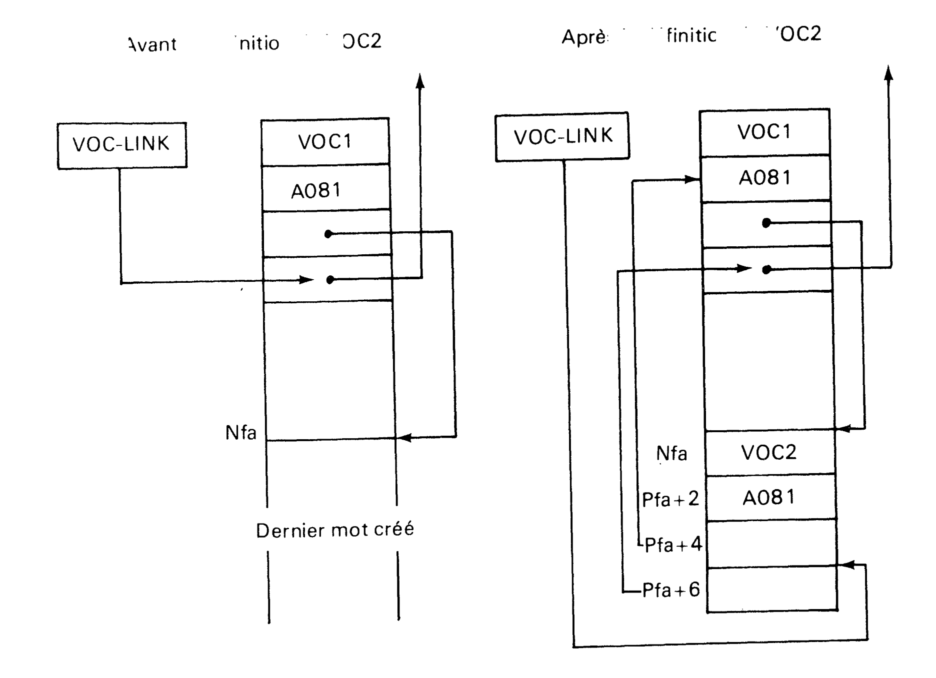
On peut représenter ces actions par le schéma suivant :



La séquence suivante :

```
HERE VOC-LINK @
VOC-LINK !
```

correspond à l'introduction de VOC2 dans le chaînage parallèle des vocabulaires.



La séquence exécutée à l'appel du nom du vocabulaire :

2+ CONTEXT !

fait pointer CONTEXT sur la cellule d'adresse PFA+4 dans la définition du vocabulaire. Le vocabulaire devient alors vocabulaire de contexte.

Si aucun mot n'a encore été défini sous ce vocabulaire, cette zone pointe sur la tête de chaîne fictive dans la définition du vocabulaire "père".

Dans le cas contraire, elle pointe sur le dernier mot défini sous le vocabulaire considéré.

Voyons en détail ce qui se passe à la création du premier mot sous VOC2, VOC2 étant alors le vocabulaire courant.

L'interpréteur va chercher dans la définition de VOC2, à l'adresse PFA+4, le NFA du dernier mot défini, qui va constituer le contenu du LFA du mot en cours de définition. Il mettra ensuite dans la cellule d'adresse PFA+4 dans la définition du VOC2 le NFA du mot qui vient d'être défini.

Ce processus général aura pour conséquence, lorsque l'on créera le premier mot de VOC2, d'initialiser le contenu du LFA de ce dernier au NFA de la tête de chaîne fictive du vocabulaire "père" de VOC2.

Il reste à voir comment passer dans un vocabulaire de définition (courant) donné : c'est le mot DEFINITIONS, dont la définition est :


```
: DEFINITIONS
        CONTEXT @
        CURRENT !
; [Return] OK
```

qui rend le vocabulaire de contexte vocabulaire courant. Ainsi, pour rendre un vocabulaire courant, il faut d'abord le rendre vocabulaire de contexte.

Exemple :

```
VOC2 DEFINITIONS [Return] OK
```

Le fait de taper VOC2 rend VOC2 le vocabulaire de contexte, et celui de taper DEFINITIONS rend le vocabulaire courant identique au vocabulaire de contexte (ici VOC2).

Dans une application FORTH professionnelle, il est crucial que l'utilisateur n'ait accès qu'à un vocabulaire restreint de mots de haut niveau, spécifiques à ses besoins.

On part de l'hypothèse que l'utilisateur n'a comme unique possibilité que de taper les mots clés de son application.

Il suffit pour cela d'isoler tous ces mots dans un même vocabulaire que l'on rend vocabulaire de contexte, puis de rompre la chaîne qui relie le premier mot de ce vocabulaire spécifique à son vocabulaire père.

L'interpréteur ne fera donc ses recherches que dans la branche ainsi isolée.

Dans un autre ordre d'idées, il est souvent utile dans des définitions de faire appel à des mots résidents dans des vocabulaires n'ayant aucun lien de parenté.

La seule manière de procéder est de rendre les vocabulaires en question immédiats. N'oubliez pas que le mot IMMEDIATE n'agit que sur le dernier mot créé. Il faudra donc rendre les vocabulaires immédiats dès leur création.

Exemple :

VOC1 et VOC2 ont été déclarés immédiats. Supposons que nous sommes dans VOC2 comme vocabulaire courant et de contexte, et que nous voulons définir un mot TEST2 qui utilise le mot TEST1 défini dans VOC1. La solution est la suivante :

```
: TEST2 ------ VOC1 TEST1 VOC2 ---;
```

. OC1 [illegible] immédiat, [illegible] rencontré l'interpréteur [illegible] rendu vocabulaire de contexte et a donc reconnu le mot TEST1. Le retour au vocabulaire de contexte initial se fait par VOC2. Ceci n'est possible que parce que les deux vocabulaires sont immédiats.

Si les deux vocabulaires n'avaient pas eu de lien direct, il aurait été nécessaire de parcourir la chaîne à la main, en repassant par FORTH par exemple.

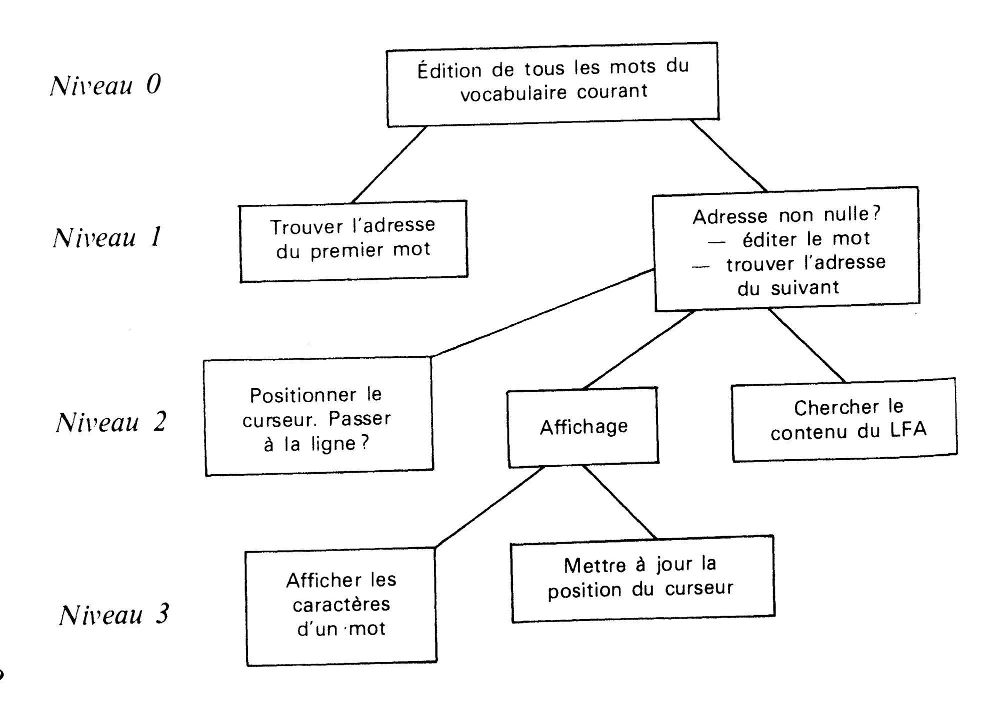
VII.2. LA SEGMENTATION

Comme le dit très bien MEYER : "La tâche de base du programmeur est de négliger à chaque niveau tous les détails non pertinents, faire abstraction de ce qui peut être remis à plus tard pour pouvoir constamment dominer les problèmes étudiés".

Chacun de ces niveaux d'abstraction représente une couche logique dans la décomposition du problème.

Exemple :

Décomposons le problème de l'affichage de tous les mots du vocabulaire courant (VLIST):



N[illegible] qu[illegible] [illegible]ntér[illegible] [illegible] cette décomposition dépend intimement de trois critères : La complexité du problème, le niveau d'évolution de l'outil de programmation et l'habileté du programmeur.

Une fois admis le principe de la décomposition d'un problème en couches logiques, une question importante se pose : Dans quel ordre aborder ces niveaux ?

Deux méthodes se présentent d'elles-même : On peut faire des décompositions successives jusqu'à aboutir à un niveau de problèmes solubles par le langage de programmation. C'est l'analyse descendante.

On peut, au contraire, partant d'outils construits briques par briques, remonter vers le problème global en s'appuyant à chaque étape sur des résultats déjà réalisés. C'est l'analyse ascendante.

Cette dernière conception n'est en fait effectivement applicable que sur la base d'une décomposition poussée des problèmes, que l'on peut regrouper par familles, pour construire des outils de plus en plus évolués.

Problèmes posés par la méthode descendante :

— Les choix initiaux, opérés aux niveaux les plus élevés sont cruciaux. Il est très difficile de les remettre en cause sans refaire totalement l'analyse.

— L'arborescence obtenue peut masquer des similitudes entre sous-arbres. C'est pourquoi il est souhaitable de toujours faire suivre une analyse descendante d'une synthèse pour mettre en lumière le caractère général de certains problèmes.

Problèmes posés par la méthode ascendante :

— L'expérience prouve qu'une synthèse ascendante mène à faire des hypothèses trop fortes dans la réalisation des différents modules.

— Les interfaces sont alors trop lourdes à manipuler. Ceci peut amener à une catastrophe lorsqu'on sait qu'en FORTH, il est surhumain de manipuler plus de quatre ou cinq données dans la pile.

— Par ailleurs, il n'est jamais exclu de trouver des incompatibilités entre problèmes à résoudre et modules déjà réalisés, ce qui amène inévitablement à remettre en cause la quasi-totalité de ces outils.

Quels enseignements tirer en FORTH de ces concepts généraux ?

FORTH fait que les définitions des mots ne peuvent dépasser une certaine taille sans entraîner une lourdeur inacceptable. Les modules qui

en...t dan. .a réa...ation d.. prob.... doi.... don. ..e co.... et r. sultants de choix délibérés.

Une des limites à la segmentation dans les langages traditionnels est la lourdeur des interfaces entre modules. (Déclarations des paramètres). FORTH résoud le problème en fournissant, en plus des variables, une pile de données.

FORTH optimise la taille occupée en mémoire par les différents modules. Il n'y a pas de passage de paramètres au sens traditionnel, la partie non-active du code correspondant à chaque module est limitée à sa plus simple expression, c'est la tête de chaîne.

On pourrait trouver FORTH limité dans les types de passages de paramètres entre modules.

Nous pensons que cette focalisation sur les passages de paramètres n'est que le fruit de mauvaises habitudes d'analyse dans des langages un peu archaïques. Ils contribuent à embrumer la lisibilité globale du programme et favorisent une certaine paresse du programmeur.

Si vous analysez sainement les problèmes, vous n'y serez jamais confronté.

Un facteur important dans la mise au point d'une méthode de programmation est celui des tests.

L'expérience prouve qu'à partir d'une certaine complexité de problèmes, la charge de travail représentée par les tests est considérable. Souvent même, le résultat s'avère imparfait, car il n'est pas toujours aisé de tester tous les cas.

Une analyse descendante par couches logiques amène naturellement à une méthode systématique de tests logiques.

Dans une couche donnée, les éléments de niveau inférieur s'ils ne sont pas écrits, sont cependant spécifiés fonctionnellement.

On peut alors les remplacer, pour tester l'élément qui les utilisera, par des échafaudages, qui effectueront des traitements simples conformes, au moins en partie, à la spécification fonctionnelle, mais sans rapport avec le problème à traiter.

Pour vous en convaincre, examinons ensemble un cas concret: Lister sur une imprimante un fichier simple en assurant correctement la gestion des en-têtes et des pieds de page.

On se fixe comme seule condition de saut de page une page pleine.

L' rith. .e n. ous . .oso. . olon....men. ...oné, ...e suivant:

```
: LISTE
      CR
      PAGE              (<— saut de page, mise à zéro du
                            compteur de ligne, édition de
      BEGIN                 l'en-tête)
          N—EOF         (<— tester la fin du fichier)
      WHILE
          LIRE          (<— lire un enregistrement)
          EDITER        (<— l'imprimer)
          INCREMENTER   (<— incrémenter le compteur de
                            lignes)
          TESTER        (<— tester le nb de lignes/page)
          IF
              F—PAGE    (<— édition du pied de page)
              PAGE
          THEN
      REPEAT
      F—PAGE
; [Return] OK
```

Les points névralgiques de cet algorithme résident dans le comportement de N—EOF et TEST.

Voici les définitions fictives nous allons donner aux autres mots, et ce sans toucher une virgule à la définition de LISTE.

```
: PAGE 0 LIGNE ! ."  EN TETE  " CR ; [Return] OK
: F—PAGE ."  PIED  " CR ; [Return] OK
: LIRE ; [Return] OK
: EDITER LIGNE @ . CR ; [Return] OK ( LIGNE contient le n° de ligne)
: INCREMENTER LIGNE 1+ ! ; [Return] OK
```

Pour tester efficacement l'algorithme, il est nécessaire d'examiner son comportement pour toutes les valeurs possibles, c'est-à-dire:

N-[illegible] vala[illegible] [illegible]mpc[illegible]ent [illegible]TEST r[illegible]us in[illegible]se pas puisque l'on ne rentre pas dans la boucle.

2) N—EOF valant 1 — TEST prenant la valeur 0 un certain nombre de fois puis prenant la valeur 1.

3) TEST valant 0 — N—EOF passant de 1 à 0.

4) TEST valant 1 — N—EOF passant de 1 à 0.

Le cas de la transition de N-EOF de 0 vers 1 est impossible du fait même de la structure des fichiers séquentiels.

cas 1

```
: N—EOF 0 ; [Return] OK

LISTE [Return]
EN TETE
PIED OK
```

On constate qu'un fichier vide déclenche une édition.
Il est toujours bon de le savoir.

cas 2

```
: N—EOF 1 ; [Return] OK
: TEST LIGNE @ 5 = ; [Return] OK

LISTE [Return]
EN TETE
1
2
3
4
PIED
ENTETE
1
.
.
OK
```

Le saut de page déclenche bien.

cas 3

```
: TEST 0 ; [Return] OK
5 VARIABLE BIDON [Return] OK
: NEOF BIDON DUP 1— +STORE @ 5 / ; [Return] OK

LISTE [Return]
EN TETE
1
2
3
4
PIED
ok
```

La fin de fichier déclenche bien.

cas 4

```
: TEST 1 ; [Return] OK

LISTE [Return]
EN TETE
1
2
3
4
PIED
EN TETE
PIED
OK
```

On a un saut de page parasite que l'on n'aurait jamais détecté sans ce crible systématique de toutes les possibilités.

C'est le cas où le saut de page dû au nombre de lignes coïncide exactement avec la fin du fichier.

L'algorithme correct est en fait le suviant:

```
''STE
    CR
    PAGE
    BEGIN
        N-EOF
    WHILE
        TESTER
        IF
            F—PAGE
            PAGE
        THEN
        LIRE
        EDITER
        INCREMENTER
    REPEAT
    F—PAGE
; [Return] OK
```

Cette méthode a le mérite, si elle est poursuivie de façon systématique, d'assurer le test couche par couche de l'ensemble de l'application.

Toutefois on a pu remarquer que la construction de tels échafaudages peut être une tâche lourde et non triviale. Une analyse poussée préalable permet dans tous les cas d'éliminer rapidement des familles de cas évidents ou impossibles.

VII.3. LA RÉCURSION

La récursion, c'est la possibilité de faire figurer dans la définition d'un objet, une référence à l'objet lui même.

Cette notion d'objet est très vaste :

Fo... m...nati... :

Fonction factorielle :

$0! = 1$

$N! = N * (N-1)! \quad \forall N>0$

Fonctions combinatoires :

$C_n^0 = 1 \qquad \forall\ n \in \mathbb{N}$

$C_n^m = 0 \qquad \forall\ m,n \in \mathbb{N}\times\mathbb{N} \text{ et } m>n$

$C_n^m = C_{n-1}^{m-1} + C_{n-1}^m \qquad \forall\ m,n \in \mathbb{N}\times\mathbb{N} \text{ et } m<=n$

— **Structures de données**

Arbre binaire :

type ARBRE-BINAIRE =(VIDE:ARBRE-BINAIRE-NON-VIDE)

type ARBRE-BINAIRE-NON-VIDE=(racine:T ; sag,sad : ARBRE-BINAIRE)

sad : sous-arbre de droite, sag : sous-arbre de gauche

Dans les deux premiers exemples, la récursion était directe, c'est-à-dire que l'appel récursif à l'objet se faisait dans la définition même de l'objet. Ce n'est pas toujours le cas, comme le montre la définition des arbres binaires.

Pour le lecteur non averti, le concept de récursion semblera peut-être n'être qu'un jeu de l'esprit dont la solution va se perdre à l'infini des appels récursifs.

On se doute bien en effet qu'une condition sine qua non dans l'utilisation de la récursion est que les objets engendrés doivent être finis. Tout ensemble de définitions récursives devra donc impérativement contenir une clause telle que dans certains cas, l'évaluation puisse se faire sans appel récursif.

Une autre condition déterminante dans le succès d'une définition récursive est la présence d'une quantité de contrôle dont on est assuré qu'elle converge strictement à chaque appel récursif.

Exemple :

— Dans le cas de la fonction factorielle, la quantité de contrôle est l'argument lui-même, et la clause d'évaluation immédiate est 0!=1.

— Dans le cas d'un arbre binaire, la quantité de contrôle est la hauteur de l'arbre, et la clause d'évaluation immédiate est VIDE.

FORTH et la récursion

FORTH n'offre à première vue aucune possibilité directe de récursion. Essayez pour vous en convaincre de définir :

```
: OBJET OBJET ; [Return]  OBJET Msg#0    OK
```

L'interpréteur n'acceptera pas cette définition car il a recherché dans le vocabulaire une définition valide des mots qui la composent, à savoir OBJET. La tête de chaîne de OBJET existe bien (créée par :), mais n'a pas encore été validée (rencontre de ;).

La solution la plus élégante consiste à valider manuellement OBJET avant l'appel récursif en utilisant le mot SMUDGE qui est,fort heureusement, immédiat.

Exemple :

```
: FACT
      SMUDGE DUP
      IF
        DUP 1 – FACT *
      ELSE
        DROP 1
      THEN
; [Return] OK
```

La définition sera acceptée cette fois par l'interpréteur. Essayez alors de l'exécuter en tapant 3 FACT par exemple.

Un message d'erreur apparaît indiquant qu'il n'existe pas de définition valide de FACT!!!.

Pour comprendre ce mystère, il faut se souvenir que SMUDGE ne fait que complémenter le SMUDGE BIT (bit de validité de définition) dans la tête de chaîne de la dernière définition créée.

Il en est de même quand ; valide une définition. Les deux complémentations successives s'annulent donc.

Il va donc suffire d'introduire une complémentation supplémentaire en intégrant un second SMUDGE en fin de définition.

La bonne définition de FACT est donc :

```
: FACT
      SMUDGE DUP
      IF
            DUP 1– FACT *
      ELSE
            DROP 1
      THEN
      SMUDGE
; [Return] OK
```

Comme nous venons de le voir, l'implantation de structures récursives directes est très aisée en FORTH.

Il existe une autre méthode plus lourde, qui consiste à réserver deux octets dans la définition du mot récursif.

On viendra ultérieurement ranger à ces emplacements réservés, le CFA du dit mot, de façon à rendre la définition définitivement récursive.

Pour ce faire, on remplace la référence récursive par un mot fictif, en l'occurence NOOP, choisi car il n'a strictement aucune action.

```
: NOOP ; [Return] OK
```

La définition étant validée, l'opérateur fait appel au mot RECURSE, qui va rechercher le CFA du dernier mot défini, et le substitue au CFA de NOOP dans la définition.

C'est ce type de méthode qui permet d'implanter des structures récursives indirectes.

Nous développerons dans le dernier chapitre deux algorithmes récursifs classiques : les tours de Hanoï, et les huits reines.

Un des obstacles à l'implantation de la récursion dans les langages évolués est le passage de paramètres lors de l'appel des procédures. Le problème qui se pose pour gérer la récursion est la sauvegarde, à chaque appel, des informations nécessaires à la poursuite de l'exécution au niveau n, après le retour de niveau n–1.

Les contextes correspondant à chaque niveau doivent donc pouvoir être rappelés dans l'ordre inverse de leur sauvegarde. On conçoit facilement que c'est typiquement une structure de pile LIFO qui permettra de gérer les sauvegardes et les rappels successifs.

C'est une des raisons pour lesquelles la plupart des langages récursifs évolués utilisent des piles, comme le fait FORTH.

VII.4. LE MULTITASKING

FORTH appartient à une famille de langages capables de gérer plusieurs tâches simultanées sans l'aide d'un système d'exploitation externe.

Dans les langages traditionnels, la résolution de ce problème a été sous-traitée au programme de contrôle de la machine, qui est généralement fourni par le constructeur.

Cette possibilité présente un intérêt considérable sur les micro-ordinateurs, où le multitasking est très rarement implanté.

Les raisons principales sont souvent de trop faibles ressources en mémoire vive pour faire résider simultanément un système d'exploitation et un langage, ainsi que la lenteur des mémoires de masse.

Avec FORTH, nous disposons des avantages suivants :

— Le noyau de base, purement réentrant est partageable par N tâches simultanées. Son taux d'utilisation est considérable, puisque l'exécution de tout mot FORTH y aboutit inévitablement.

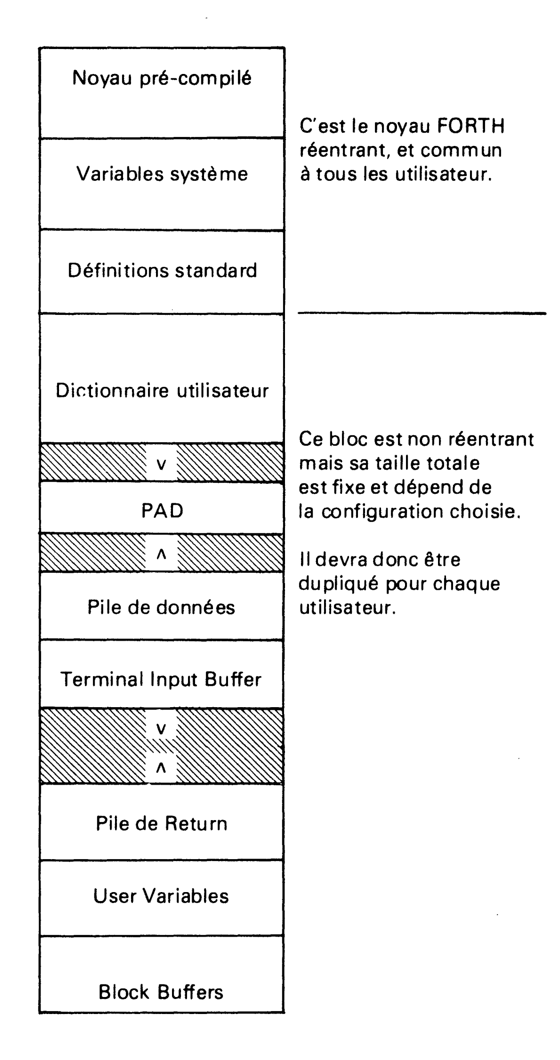
— L'allocation mémoire pour chaque tâche est dynamique : tous les mots qui composent les différentes applications sont rangés séquentiellement en mémoire, et sont reliés par vocabulaires grâce au chaînage entretenu par l'interpréteur. De plus, les définitions de mots sont extrêmement compactes. Une application FORTH complète et bien écrite occupe moins de place que si elle avait été écrite en ASSEMBLEUR.

Il est tout à fait raisonnable d'imaginer cinq tâches travaillant simultanément en FORTH sur 48 Koctets, ce qui correspond à une configuration classique de micro-ordinateur 8 bits.

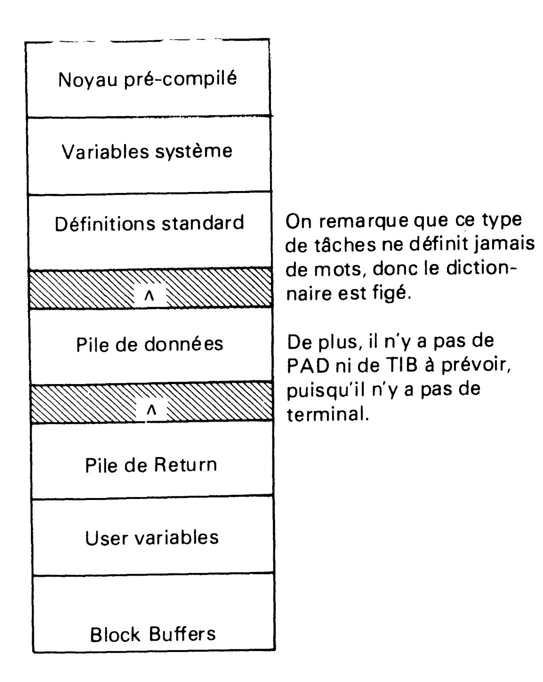
L'objet de ce paragraphe n'est en aucun cas d'entrer dans le détail de l'implantation du multitasking. La gestion des ressources non partageables, par exemple, est, selon la formule trop bien connue, "beyond the scope of this book". Nous nous proposons simplement de dégager les grands principes d'implantation.

Il existe un "super FORTH" appelé Poly-FORTH qui propose entre autre le multitasking et qui est commercialisé pour de nombreuses machines.

L'examen de la memory map d'un FORTH mono utilisateur amène les remarques suivantes :



Dans un système multi-utilisateurs, on aura besoin de tâches secondaires, fonctionnant en back-ground. La memory map de telles tâches pourra avoir la structure suivante :



Pour pouvoir configurer dynamiquement la taille des zones mémoire allouées à chaque tâche, il sera pratique de particulariser un utilisateur, qui jouera le rôle d'opérateur.

La configuration la plus naturelle pour le système multi-utilisateur sera donc la suivante:

Noyau pré-compilé
Variables système
Définitions standard
Utilisateur 1
Utilisateur 2
Utilisateur 3
Contrôle 1
Contrôle 2
Opérateur
Block Buffers

Les Block Buffers sont partagés pour faciliter la gestion du partage de la mémoire de masse.

RÉSUMÉ DU CHAPITRE VII

Mot	*Syntaxe*	*Définition*
VOCABULARY	VOCABULARY "voc"	Mot de définition d'un vocabulaire dont le nom est "voc".
DEFINITIONS	"voc" DEFINITIONS	Le vocabulaire de contexte étant "voc" (dès que l'on tape voc), le vocabulaire courant devient aussi "voc".
SMUDGE	SMUDGE	Complémente le smudge bit du dernier mot créé.

VIII
PROBLÈMES

Dans ce chapitre, nous vous proposons de résoudre en FORTH quelques problèmes simples, pour vous familiariser avec le langage.

Les problèmes sélectionnés sont les suivants :

1. Manipulations de nombres complexes.
2. Le jeu de la Vie (Univers de CONWAY).
3. Fonctions trigonométriques.
4. Les tours de HANOI (Récursion).
5. Les huit Reines (Récursion).
6. Le calendrier grégorien perpétuel.
7. Fichiers séquentiels.
8. La transformée de Fourier discrète.

PROBLÈME 1 — MANIPULATIONS DE NOMBRES COMPLEXES

Rappels mathématiques.

Un nombre "complexe" est en fait un couple de nombre réels sur lequel on autorise les opérations suivantes :

c1 := (a1,b1)   c2 := (a2,b2)

addition interne : $c1+c2 := (a1+a2,b1+b2)$

multiplication externe par un nom réel : $s.c1 := (s.a1,s.b1)$

multiplication interne par un nombre complexe : $c1*c2 := (a1.a2 - b1.b2,a1.b2+a2.b1)$

Il ne saurait être question d'étendre plus avant l'ensemble des propriétés mathématiques de cet ensemble. Notons simplement que l'on convient de représenter un nombre complexe sous la forme :

$c := a+ib$

où "i", nombre complexe par excellence, vérifie :

$i*i = -1$

L'ensemble des nombres complexes, espace vectoriel de dimension deux, admet donc pour base 1 et i.

On convient également d'appeler "conjugué" d'un nombre complexe son symétrique par rapport à l'axe réel.

Ainsi : $c := a+ib$

aura pour conjugué : $c' := a-ib$

On appelle également "module" d'un nombre complexe la grandeur :

$|c| := (a.a+b.b)^{1/2}$

Ce qui géométriquement représente la longueur du vecteur réel :

$v = (a,b)$

Compte tenu de ce que $i*i=-1$, on constate que $|c|^2 = c*c'$.

Pour continuer avec les analogies géométriques, on introduit naturellement "l'argument" d'un nombre complexe, qui correspond à l'angle du vecteur réel avec le vecteur (1,0) :

$arg(c) = arctg\ (b/a)$

On pourra donc écrire :

$c := |c|.(\ sin\ (arg(c)) + i\ cos\ (arg(c))\ )$

On peut remarquer les propriétés suivantes des fonctions module et "argument" :

$|c1*c2| = |c1|.|c2|$

$|c1+c2| \leqslant |c1| + |c2|$

$arg(c1*c2) = arg(c1) + arg(c2)$

Cette dernière propriété est intéressante, car elle va nous permettre d'étendre la notion d'exponentielle réelle à celle d'exponentielle complexe, puisque l'on convient de noter :

$c = d.(sin(x)+i.cos(x)) = d\ exp(ix)$

où $d = |c|$ et $x = arg(c)$.

> Le problème consiste à générer la structure de donnée "nombre complexe", en virgule fixe, avec quelques opérateurs d'application, à savoir :
>
> — addition, soustraction,
> — multiplication par un réel,
> — multiplication de deux nombres complexes,
> — module,
> — argument.

1. Génération de nombres complexes

Comme vous vous en doutez, il faut utiliser les outils de définition de mots de définition pour résoudre cette première étape.

La nouvelle structure de données qui nous intéresse présente les caractéristiques suivantes :

— Chaque donnée comprend simplement deux nombres réels, du fait de l'isomorphisme du plan réel et du plan complexe.

— L'accès à chaque variable complexe se fait par son nom, ce qui aura pour effet de retourner sur la pile l'adresse de la composante imaginaire, la composante réelle la suivra en mémoire.

— La syntaxe de création d'une variable complexe étant :

```
PR IM COMPLEXE nom-de-la-variable
```

où PR représente la partie réelle, et IM la partie imaginaire pure.

La définition du mot de définition de variables complexes sera donc, de façon naturelle :

```
COMPLEXE
      <BUILDS
                    (range successivement dans le dictionnaire)
      , ,           (les deux valeurs au sommet de la pile.)
      DOES>
                    (l'adresse souhaitée est déjà sur la pile)
;
Return OK
```

On pourra maintenant écrire :

```
1 2 COMPLEXE UN—DEUX Return OK
```

La structure ainsi générée a donc l'anatomie suivante :

Tête de chaîne
pointeur
pointeur
2
1

Dotons-nous immédiatement d'outil pour accéder à la valeur des variables complexes :

```
: X@            ( adr ... re im )
    DUP         ( adr adr ... )
    @           ( adr im ... )
    SWAP        ( im adr ... )
    2+          ( im adr+2 ... )
    @           ( im re ... )
    SWAP        ( re im ... )
; Return OK

: X!            ( re im adr ... )
    SWAP   OVER   !   2+  !
; Return OK
```


Opérateurs pour nombres complexes

L'addition se fait de façon très simple :

```
: X+      ( re1 im1 re2 im2 ... re1+re2 im1+im2 )
    ROT   ( re1 re2 im2 im1 ... )
    +     ( re1 re2 im1+im2 ... )
    >R    ( re1 re2 ... )
    +     ( re1+re2 ... )
    R>    ( re1+re2 im1+im2 ... )
; Return OK
```

La soustraction :

```
: X—      ( re1 im1 re2 im2 ... re1—re2 im1—im2 )
    ROT   ( re1 re2 im2 im1 ... )
    SWAP  ( re1 re2 im1 im2 ... )
    —     ( re1 re2 im1—im2 ... )
    >R    ( re1 re2 ... )
    —     ( re1—re2 ... )
    R>    ( re1—re2 im1—im2 )
; Return OK
```

La multiplication par un réel,

```
: X*'             ( re im sc ... re*sc im*sc )
    SWAP OVER     ( re sc im sc ... )
    * ROT ROT *   ( im*sc re*sc ... )
    SWAP          ( re*sc im*sc ... )
; Return OK
```

Enfin la multiplication de deux nombres complexes :

```
X*            ( r... r2 i... 1.r2 ... 2 r... +re[illegible] )
2OVER 2OVER   ( C1 C2 C1 C2 ... )
  ROT *       ( C1 C2 r1 r2 i1.i2 ... )
  R> * R> -   ( C1 C2 r1.r2-i1.i2 ... )
  R>          ( r1 r2 i1 i2 ... )
  ROT ROT *   ( r1 i2 i1.r2 ... )
  R> * R> +   ( r1.i2 i1.r2 ... )
  R>          ( r2.i2+i1.r2 r1.r2-i1.i2 ... )
; Return OK
```

Le conjugué d'un nombre complexe est pour mémoire:

```
: BAR MINUS ; Return OK
```

On ne peut alors accéder au module et à l'argument que si l'on a la fonction racine.

```
: R2 2OVER BAR X* ; Return OK

: ARG         ( re im ... sinus(argument) cosinus(argument))
    2DUP      ( re im re im ... )
    R2 SQR    ( re im d ... )
    1 SWAP/   ( re im 1/d ... )
    X*
; Return OK
```

PROBLÈME 2 — LE JEU DE LA VIE

Les règles de ce jeu furent pour la première fois exposées à l'occasion d'un article publié par Martin GARDNER en 1970 dans le Scientific American. La paternité en revient à John CONWAY, de l'université de Cambridge.

Le principe est d'obtenir une population d'objets dans un univers plan qui vont naître, vivre, se reproduire et mourir selon des règles sociales très simples.

Les critères de choix de ces règles sociales sont les suivants:

1. Il ne doit pas exister de forme initiale telle qu'une démonstration simple indique une croissance illimitée.

2. Il doit exister des formes initiales qui entraînent une croissance illimitée.
3. Il doit exister des formes initiales qui changent pendant une très longue période de temps avant de finir de l'une des trois façon suivantes:

— disparition complète,
— forme stable,
— oscillation sur plus de deux périodes.

En bref, l'évolution de la population ne doit pas être prévisible.

L'environnement de chaque cellule est constitué par les huit cellules adjacentes (quatre perpendiculaires et quatre diagonales).

Les lois sont les suivantes:

— Toute cellule avec deux ou trois voisins survit.

— Toute cellule ayant au moins 4 voisins meurt de surpopulation, et toute cellule ayant moins de 2 voisins meurt d'isolement.

— Tout emplacement libre ayant trois voisins est l'objet d'un heureux événement.

Les naissances et les morts ont lieu simultanément, et correspondent ensemble à une nouvelle génération.

Réalisation:

Pour des raisons évidentes, nous allons restreindre le champ d'évolution de la société à un plan fermé de dimension finie.

Il est naturel de représenter notre univers par un tableau, où la valeur de chaque cellule indique la présence ou l'absence d'un habitant.

Pour visualiser les évolutions, on affichera le tableau en affichant des étoiles aux emplacements correspondants aux cellules habitées, et des espaces blancs partout ailleurs.

Le problème se pose de ne pas faire interférer dans l'algorithme les naissances et les morts d'une même génération. Il faut donc balayer tout le contenu du tableau, localiser les cellules sujettes à modification, et effectuer toutes les modifications simultanément, pour pouvoir construire la génération suivante.

pre[illegible] idée [illegible] vien[illegible] esprit [illegible] de travailler sur deux tableaux. Cette solution, bien que simple, est forte consommatrice de mémoire.

Une solution plus élégante consistera à coder dans chaque cellule d'un même tableau non seulement la présence ou l'absence d'un élément à la génération N, mais aussi des informations sur son avenir à la génération N+1.

Le codage choisi est :

Génération N	Génération N+1
Existant : 1	Mort : 5 Survie : 1 ou 3
Inexistant : 0	Naissance : 2 Inchangé : 0 ou 4

En d'autres termes, une cellule sera occupée à la génération N+1 si la valeur de cette même cellule à l'issue du balayage de la génération N est 1, 2 ou 3. Dans tous les autres cas, la cellule sera vide.

On peut d'ores et déjà écrire les grandes lignes de l'algorithme, le nombre d'étapes étant placé sur la pile avant l'exécution.

```
: JEU 0 DO
        AFFICHER        ( Affiche la génération N )
        GENERER         ( Balaye la génération N )
        NORMALISER      ( Construire la génération N+1 )
    LOOP
    AFFICHER            ( Affiche la dernière génération )
;
```


Il re[illegible] donc [illegible] définir les mots A[illegible]ICHER, GENERER et NORMALISER, ainsi que les mots :

— CLEAR qui permettra une remise à zéro de l'univers.

— SET et RESET qui permettront d'initialiser la population.

L'outil de base dans la définition de tous ces mots est le mot de définition ARRAY qui permettra de manipuler aisément le tableau.

Pour définir le tableau univers, on écrira :

```
DimensionX DimensionY ARRAY UNIVERS
```

et pour accéder à une cellule (X, Y) on tapera :

```
X Y UNIVERS
```

qui retournera sur la pile l'adresse de la cellule correspondante.

La définition de ARRAY sera :

```
: ARRAY                 ( DimX DimY --- )
    <BUILDS
        OVER ,          ( Met DimX au Pfa+2 )
        * ALLOT         ( Réserve DimX * DimY octets )
    DOES>
        DUP @           ( Ramène DimX sur la pile )
        ROT *           ( Calcul de IndY * DimX )
        + + 2+          ( Calcul adresse absolu de la cellule )
```

On remarquera que les indices des tableaux générés par ARRAY évoluent dans l'intervalle [0,DimX−1] * [0,DimY−1].

Il est maintenant très aisé de définir les mots suivants :

```
:T                                  (        IndY      - )
    UNIVERS   1 SWAP   C!
;
: RESET                             ( IndX IndY ——— )
    UNIVERS   0 SWAP   C!
;
: CLEAR                             ( ——— )
    DIMX   0   DO
        DIMY   0   DO
            I   J   RESET
        LOOP
    LOOP
;
```

DIMX et DIMY sont deux constantes préalablement définies qui donnent les dimensions du tableau.

Pour l'affichage, on écrira :

```
: AFFICHAGE                         ( ——— )
    HOMEUP                          ( Efface l'écran )
    DIMY   0   DO
        DIMX   0   DO
            I   J   UNIVERS   C@
            IF
                ." * "
            ELSE
                SPACE
            THEN
        LOOP
        CR
    LOOP
;
```


L'alg… ime … orma… ion, … pte t… de la …ification choi… est très simple :

```
: NORMALISER                                    ( ——— )
    DIMY   0   DO
        DIMX   0   DO
            I   J   UNIVERS   DUP   C@   ( Ad val ——— )
            DUP   3>
            IF
                DROP   0                 ( Ad   0 ——— )
            ELSE
                DUP   1>
                IF
                    DROP   1             ( Ad   1 ——— )
                THEN
            THEN
            SWAP   C!                    ( Rangement nouvelle
                                           val )
        LOOP
    LOOP
;
```

Le cœur de l'algorithme réside dans le mot GENERER. On peut le décomposer en plusieurs blocs fonctionnels :

1 — Balayer tous les éléments du tableau.

2 — Pour chaque élément, isoler le sous-tableau (3*3) qui l'entoure, en tenant compte de la présence éventuelle des bords du tableau.

3 — Totaliser les éléments présents dans ce sous-tableau à la génération N, dont la valeur est 1, 3 ou 5, c'est-à-dire dont le bit de poids faible est positionné.

4 — Retirer éventuellement 1 si l'élément en cours d'examen existe à la génération N.

5 — Encoder en conséquence la valeur de l'élément considéré.

```
-- VER--
  DIMY  0  DO
     DIMX  0  DO
        0                              ( Compteur voisins )
        J  2+  DIMY  MIN
        I  1  -  0  MAX
        DO
           J  2+  DIMX  MIN
           I  1  -  0  MAX
           DO
              I  J  UNIVERS @
              1  AND  +
           LOOP
        LOOP
        I  J  UNIVERS @
        1  AND                         ( Nbvois
                                         Present --- )
        SWAP  OVER  -
        VERIFIER                       ( Encodage de
                                         l'avenir )
        I  J  UNIVERS  !
     LOOP
  LOOP
```

Présent est un booléen indiquant si l'élément examiné est habité ou non à la génération N. Il est déterminant pour l'avenir de la cellule considérée.

VERIFIER comprend tout l'encodage de l'élément en fonction du nombre de voisins, et de l'état vide ou occupé de la cellule à la génération N.


```
. VERIFIER                             ( État Nbvois --- )
     DUP  3  =                         ( État Nbvois b --- )
     IF
        DROP  2+                       ( État+2 --- )
     ELSE
        2  =  0=                       ( État État≠2 --- )
        IF
           4  +
        THEN
     THEN
;
```

La définition ci-dessus résulte du tableau récapitulatif suivant :

N | N + 1

	V=0 V=1	V=2	V=3	V>3
État=1 Occupé	État=État+4=5 (mort par isolement)	Inchangé	État=État+2=3 (survie)	État=État+4=5
État=0 Vide	État=État+4=4 (inchangé)	Inchangé	État=État+2=2 (naissance)	État=État+4=4 (inchangé)

PROBLÈME 3 — FONCTIONS TRIGONOMÉTRIQUES

> Le problème consiste à construire les fonctions trigonométriques classiques, en virgule fixe, à savoir SINUS et COSINUS.

Ces fonctions étant continues, bornées, et périodiques, il est très facile de générer une table des valeurs de ces fonctions.

Le pas sera décidé à l'avance en fonction de la précision accordée au calcul, c'est-à-dire de la position de la virgule.

Nous avons choisi arbitrairement une précision de 4 chiffres.

... défi...ns s... alors ...médi... .

```
: TABLE
    <BUILDS
    0 DO , LOOP
    DOES>
    SWAP 2 * + @
;

10000 9998 9994 9986 9976 9962 9945 9925 9903 9877
 9848 9816 9781 9744 9703 9659 9613 9563 9511 9455
 9397 9336 9272 9205 9135 9063 8988 8910 8829 8746
 8600 8572 8480 8387 8290 8192 8090 7986 7880 7771
 7660 7547 7431 7314 7193 7071 6947 6820 6691 6561
 6428 6293 6157 6018 5878 5736 5592 5446 5299 5150
 5000 4848 4695 4540 4384 4226 4067 3907 3746 3584
 3420 3256 3090 2924 2756 2588 2419 2250 2079 1908
 1736 1564 1392 1219 1045 0872 0698 0523 0349 0175
 0000
91 TABLE SINTABLE  [Return] OK
```

Le mot TABLE crée et remplit effectivement une table dont le nombre d'entrées est paramétré.

Ici, SINTABLE contient 91 valeurs de la table des sinus, entre 0 et 90 degrés.

Pour pouvoir accéder à cette table pour toutes les valeurs d'angles possibles, il convient de prendre quelques précautions :

```
: S180
    DUP 90 >
    IF
        180 SWAP -
    THEN
    SINTABLE
; [Return] OK
```


a al

```
: SIN                                  ( arg ... sinus(arg) )
    360 MOD
    DUP 0<
    IF 360 + THEN
    DUP 180 >
    IF 180 - S180 MINUS ELSE S180 THEN
; [Return] OK

: COS 90 + SIN ; Return ok
```

On peut maintenant écrire quelques outils utilisant des fonctions trigonométriques :

```
: ERRMATH ." VALEUR TROP GRANDE" QUIT ; [Return] OK
: TANG                          ( arg ... tg(arg) )
    DUP SIN
    SWAP COS
    DUP IF / ELSE DROP ERRMATH THEN
; [Return] OK
```

PROBLÈME 4 — LES TOURS DE HANOI

> Problème frivole, mais digne d'intérêt.
>
> Les prêtres d'une secte orientale ont à résoudre le problème suivant :
>
> 64disques sont disposés les uns sur les autres, sur un socle "A", les diamètres des disques devant toujours aller décroissants. Le problème consiste à transférer tous ces disques vers un socle "C", en utilisant un socle intermédiaire "B".
>
> Les seuls mouvements autorisés sont ceux qui consistent à prendre un disque au sommet d'une des piles, et à le poser soit sur un disque plus grand, soit sur un socle vide.

Selon la légende, la fin du monde surviendra à l'instant où les moines auront terminé leur labeur.

Pour aider les moines à parvenir à leur fin, nous allons leur éditer un listing de la suite des mouvements de disques à opérer.

Tout le problème consiste donc à bâtir un algorithme dérivant toutes les etapes du transfert.

Il faut savoir que cet exercice cache un prototype d'une classe importante d'algorithmes récursifs, car si nous essayons la démarche qui consiste à toujours se ramener au problème précédent, nous pouvons écrire :

Déplacer 64 disques de "A" vers "C", c'est :

```
Déplacer 63 disques de "A" vers "B"
IMPRIMER "A" vers "C"
Déplacer 63 disques de "B" vers "C"
```

De proche en proche, le problème va se simplifier, jusqu'à ce qu'il n'y ait plus de plateaux sur un des socles, auquel cas, le problème "Déplacer 0 plateau" sera instantanément résolu.

On peut donc écrire une définition complète de "Déplacer" :

Déplacer N disques de DEPART vers ARRIVEE :

```
SI N≠0 ALORS
   Déplacer N−1 disques de DEPART vers INTERMEDIAIRE
   IMPRIMER DEPART "VERS" ARRIVEE
   Déplacer N−1 disques de INTERMEDIAIRE vers ARRIVEE
```

Pour paramétrer suffisamment le problème, on numérotera les socles 1, 2 et 3.

A tout instant, on profite de la relation :

```
DEPART  +  INTERMEDIAIRE  +  ARRIVEE = 6
```

on va donc écrire en FORTH :

```
: HANOI                    ( d a n ... )
       SMUDGE
       DUP
       IF                  ( d a n ... )
         PREPARE−ALLER     ( d a n   d i n−1 ... )
         HANOI             ( d a n ... )
         EDITE
         PREPARE−RETOUR    ( d a n i a   n−1 ... )
         HANOI             ( d a n ... )
       THEN
       DROP DROP DROP
  SMUDGE
; Return OK
```


Les mots PREPARE préparent la pile pour l'appel suivant :

```
: EDITE
       3DUP DROP SWAP CR .  ."  VERS" .
; Return OK

: PREPARE−ALLER              ( d a n ... d a n d i n−1 )
       3DUP                  ( d a n d a n ... )
       ROT  ROT  OVER  +     ( d a n n d a d ... )
       6 SWAP  −  ROT  1  −  ( d a n d i   n−1 ... )
; Return OK

: PREPARE−RETOUR             ( d a n ... d a n i a   n−1 )
       3 DUP                 ( d a n d a n ... )
       SWAP ROT OVER +       ( d a n n a d a ... )
       6 SWAP − SWAP ROT 1 − ( d a n i a   n−1 ... )
; Return OK

: 3DUP ( n1 n2 n3 ... n1 n2 n3 n1 n2 n3 )
       DUP >R ROT
       DUP >R ROT
       DUP >R ROT
       R> R> SWAP R>
; Return OK
```

PROBLÈME 5 — LES HUITS REINES

> Autre problème frivole, mais moins immédiat que le précédent.
>
> Le but est de construire un mot qui affichera à l'écran de votre terminal une combinaison possible de huit reines sur un échiquier, sans qu'aucune d'elles ne soit en prise.

Si vous aviez à le faire réellement, vous auriez sans doute le réflexe de

raisonnant par lignes ou colonnes, en faisant des déplacements rangée par rangée.

On peut formaliser cette solution de la façon suivante:

Pour placer N reines sur l'échiquier, (8–N sont déjà posées).

Trouver la première ligne disponible pour une reine.
Trouver la première colonne disponible pour une reine.
Placer la reine à cet emplacement.
Placer N–1 reines sur l'échiquier.

Il faudra bien entendu traiter le cas trivial, où il ne reste plus de reines à placer.

Le mot FORTH qui va résoudre le problème va s'écrire de façon très simple:

```
: HUIT–REINES ( ... )
      INITIALISER
      8 PLACER
; [Return] OK
```

Le mot INITIALISER va mettre à zéro la variable qui va garder à tout instant la carte des emplacements disponibles.

```
: INITIALISER ( ... )
    8  0 DO
       8  0 DO
          0 I J SET
       LOOP
    LOOP
; [Return] OK
```

Les valeurs vont être stockées dans un tableau ECHIQUIER, et on y accèdera grâce aux mots SET et TEST.

```
0  VARIABLE ECHIQUIER 64 ALLOT [Return]  OK

: SET  8 *  +  ECHIQUIER  +  C! ; [Return]  OK  ( val col lig ... )
: TEST 2 DUP 8 * + ECHIQUIER + C@ ; [Return]  OK ( col lig ...
                                                   ... col lig val )
```

Les fonctions de "Trouver ligne" et "Trouver colonne" sont assez semblables, mais seront différenciées pour plus de clarté.


```
: TROUVER–LIGNE        ( ... ligne )
    0 0                ( 0 0 ... )
    BEGIN              ( 0 ligne ... )
     TEST              ( 0 ligne ... 0 ligne flag )
    WHILE              ( 0 ligne ... )
     1+                ( 0 ligne+1 ... )
    REPEAT
    SWAP DROP          ( ligne ... )
; [Return] OK

: TROUVER–COLONNE      ( ... col )
    0 0                ( 0 0 ... )
    BEGIN              ( col 0 ... )
       SWAP TEST       ( 0 colonne ... 0 colonne flag )
    WHILE              ( 0 colonne ... )
       1+              ( 0 colonne+1 ... )
       SWAP            ( colonne+1 0 ... )
    REPEAT
    DROP               ( col ... )
; [Return] OK
```

Il reste donc à écrire un mot qui va mettre à jour l'échiquier après avoir positionné une reine.

```
JPD                    (      g ...
     SET–COL
     SET–LIG
     SET–DIAG
     DROP DROP
; [Return] OK
```

Avec :

```
: SET–COL          ( col lig ... col lig )
     SWAP          ( lig col ... )
     8 0 DO
       0 OVER I SET
     LOOP
     SWAP          ( col lig ... )
; [Return] OK
```

De même pour :

```
: SET–LIG                      ( col lig ... col lig )
     8 0 DO                    ( col lig ... )
        1 OVER I SWAP SET      ( col lig ... )
     LOOP
; [Return] OK
```



```
: SET–DIAG                        ( col lig ... col lig )
     2DUP SWAP                    ( col lig lig col ... )
   – DUP 9 + SWAP                 ( col lig lig–COL+9 lig–col ... )
   DO                             ( col lig ... )
     2DUP SWAP – I SWAP I +       ( col lig I I+lig–col ... )
     1 ROT ROT                    ( col lig 1 I I+lig–col ... )
     SET                          ( col lig ... )
   LOOP
     2DUP 1 + + 0                 ( col lig col+lig+1 0 ... )
   DO                             ( col lig ... )
     2DUP + SWAP OVER –           ( col lig I col+lig–I ... )
     1 ROT ROT                    ( col lig ... )
     SET
   LOOP
;
[Return] OK
```

On peut donc maintenant très facilement écrire le mot PLACER :

```
: PLACER                          ( N ... )
     SMUDGE
     DUP IF
          TROUVER–COLONNE         ( N col ... )
          TROUVER–LIGNE           ( N col lig ... )
          UPDATE                  ( N col lig ... )
          AFFICHE                 ( N ... )
          1 –                     ( N–1 ... )
          PLACER                  ( N–1 ... )
     ELSE
          DROP EFFACE
     THEN
; [Return] OK
```

AFFICHE [illegible] EFFACE sont [illegible] m[illegible] [illegible] vo[illegible] [illegible] endr[illegible] cha[illegible] visualisation des résultats.

Nous vous proposons un affichage très simplifié:

```
: AFFICHE ."  LIGNE:" .  ."  COLONNE" .; [Return] OK

: EFFACE
     10 EMIT 10 EMIT        ( sauter deux lignes )
     ."  TERMINE"
     KEY  DROP
; [Return]  OK
```

PROBLEME 6 — LE CALENDRIER GRÉGORIEN PERPÉTUEL

> Le problème consiste à trouver, étant donnée une date, le jour de la semaine correspondant.

Une analyse détaillée du problème a conduit à l'élaboration des trois formules suivantes, valables seulement à partir de l'an de grâce 1582.

Pour les mois de janvier à février:

```
Facteur =   365 * Année
          + Jour
          + 31 * ( Mois -1 )
          - INT ( 3/4 ( INT ( Année / 100 ) + 1 ) )
          + INT ( ( Année - 1 ) / 4 )
```

Pour les mois de mars à décembre:

```
Facteur =   365 * Année
          + Jour
          + 31 * ( Mois - 1 )
          - INT ( 3/4 ( INT ( Année / 100 ) + 1 ) )
          - INT ( 0.4 * Mois + 2.3 )
          + INT ( Année / 4 )
```

La formule finale est alors:

N° Jour dans la semaine = Facteur MOD 7

(Le samedi étant le jour de N° 0.)

On remarque que seule une petite partie des deux formules diffère, d'où les deux mots suivants:

```
: JANVIER/FEVRIER        ( Année Mois --- n )
    DROP  1  -           ( Année-1 --- )
    4  /                 ( INT((Année-1)/4) --- )
; [Return] OK

: MARS/DECEMBRE          ( Année Mois --- n )
    4  *  23  +  10  /   ( Année INT(0.4*Mois+2.3) --- )
    SWAP  4  /           ( INT(0.4*Mois+2.3) INT(Année/4) --- )
    SWAP  -
; [Return] OK
```

Voici maintenant un mot qui affiche le jour de la semaine en fonction de son n°.

```
: EDITE                                      ( N° Jour --- )
    DUP 0=
    IF  ."  SAMEDI"
    ELSE
       DUP 1 =
       IF  ."  DIMANCHE"
       ELSE
          DUP 2 =
          IF  ."  LUNDI"
          ELSE
             DUP 3 =
             IF  ."  MARDI"
             ELSE
                DUP 4 =
                IF  ."  MERCREDI"
                ELSE
                   DUP 5 =
                   IF  ."  JEUDI"
                   ELSE
                      DUP 6 =
                      IF  ."  VENDREDI"
    THEN THEN THEN THEN THEN THEN THEN
    DROP
    CR
; [Return] OK
```

Certains résultats intermédiaires étant en double longueur, nous allons utiliser dans la définition des mots qui suivent des opérateurs mixtes.

Nous considérons que la date sera accessible sur la pile de donnée, sous la forme :

```
                              ( Jour Mois Année --- )

: JOUR                        ( J M A --- )
   DUP  365  M*               ( J M A d --- )
   >R  >R  ROT                ( M A J --- )
   0  R>  R>  D+              ( M A d --- )
   2SWAP  2DUP  >R >R         ( d M A --- )
   100  /  1+  3  4  */       ( d M n --- )
   SWAP  1  -  31  *          ( d n n' --- )
   SWAP  -  0  D+             ( d --- )
   R>  R>                     ( d M A --- )
   SWAP  DUP  3<              ( d A M flag --- )
   IF
      JANVIER/FEVRIER
   ELSE
      MARS/DECEMBRE
   THEN                       ( d n --- )
   0  D+                      ( Facteur --- )
   7 MOD                      ( N° Jour --- )
   EDITE                      ( --- )
; Return OK
```

Vous pouvez maintenant taper par exemple :

```
7 12 1982 JOUR Return MARDI
OK
```

PROBLÈME 7 — FICHIERS

> Le but de ce problème est de pouvoir créer et se servir d'un fichier.
>
> Les informations que nous utiliserons serons du type : Nom Prénom Téléphone Société.
>
> Chacune de ces séquences sera appelée enregistrement, et chaque classe d'information dans un enregistrement un champ.
>
> Nous donnerons les trois mots qui permettront les manipulations de base : écriture, modification et lecture. Comme nous le verrons d'autres mots pourront être très simplement écrits.

Nous allons utiliser la structure de mémoire de masse fournie par FORTH, principalement le mot BLOCK.

Commençons par définir quatre variables qui donneront les caractéristiques du fichier :

— LE qui est la longueur d'un enregistrement. Nous prendrons dans cet exemple 64 caractères.

— PB qui est le numéro du premier bloc de la mémoire de masse à servir pour le fichier.

— EB qui est le nombre d'enregistrements contenus dans un bloc, c'est-à-dire le nombre d'octets par bloc divisé par la longueur d'un enregistrement (LE).

— CM qui est le nombre d'enregistrements maximum du fichier : il est naturel de choisir un multiple de EB.

Les blocs constituant le fichier sont ceux qui suivent séquentiellement PB. Il est donc important de bien choisir PB, c'est-à-dire un numéro de bloc correspondant bien à une série de blocs libres.

Nous aurons également besoin de cinq variables :

— ENRG qui sera le pointeur d'enregistrement, c'est-à-dire le numéro de l'enregistrement à traiter.

— RECH qui contiendra l'adresse d'une table de référence pour une recherche.

— RECH1 qui servira pour une exécution vectorisée.

— RECH2 qui servira pour une exécution vectorisée.

— DEB qui sera le numéro de l'enregistrement à partir duquel une recherche sera effectuée.

Nous aurons aussi besoin de quatre tables à deux éléments, chacune correspondant à un champ et contenant la longueur du champ ainsi que sa position dans l'enregistrement (tabulation). Pour ce faire nous avons besoin du mot de définition suivant :

```
: DEF
      <BUILDS  ,  ,
      DOES>
; Return OK
```

Les tables seront donc définies de la façon suivante :

```
1[illegible] DEF [illegible]M
12 16 DEF PRENOM
13 28 DEF TELEPHONE
23 41 DEF SOCIETE
```

Dans un enregistrement le nombre maximum de caractères pour le nom est de 16, pour le prénom de 12, pour le téléphone de 13 et pour le nom de la société de 23.

Tout d'abord écrivons un mot DEBUT qui initialise à zéro le numéro d'enregistrement et un mot SUITE qui l'incrémente de un :

```
: DEBUT
      0 ENRG !
; [Return] OK
: SUITE
      1 ENRG +!
; [Return] OK
```

Nous allons maintenant écrire un mot ENRG—>ADR qui à partir du contenu de ENRG va calculer dans quel bloc est situé cet enregistrement, charger ce bloc en mémoire s'il n'y est déjà et laisser sur la pile l'adresse mémoire à partir de laquelle est situé cet enregistrement.

```
: ENRG—>ADR                      ( ——— Adr)
         ENRG @   EB /MOD        (Incr N°bloc ——— )
         PB   4   *   +   BLOCK  (Incr Adr ——— )
         SWAP LE * +             ( Adr ——— )
;  [Return]  OK
```

L'étape suivante est d'écrire un mot qui ayant sur la pile l'adresse de la table de l'un des champs (NOM, PRENOM, TELEPHONE, SOCIETE), laisse sur la pile l'adresse de ce champ dans l'enregistrement pointé (par ENRG) ainsi que le nombre de caractères qui le constituent (16,12,13,23).

```
: ?CHAMP                         (Champ ——— Adr   Long)•
      DUP @  SWAP   2 + @ SWAP   (Long Tab ———)
      ENRG—>ADR                  (Long Tab Adr ———)
      +   SWAP                   (Adr Long ———)
; [Return] OK
```


Il est maintenant très simple d'écrire un mot .CHAMP qui ayant l'adresse du champ sur la pile affichera le contenu du champ de l'enregistrement pointé par ENRG :

```
: .CHAMP          (Champ ———)
      ?CHAMP      (Adr Long ———)
      TYPE
; [Return] OK
```

De même il est très simple d'écrire le mot CHAMP! qui ayant l'adresse du champ sur la pile enregistrera le texte tapé dans le champ choisi de l'enregistrement pointé par ENRG :

```
: CHAMP!                                 (Champ ———)
         44 TEXT PAD SWAP                (AdrPAD  Champ ———)
         ?CHAMP CMOVE UPDATE
; [Return] OK
```

Le lecteur remarquera que 44 est le code ASCII de la virgule qui a été choisie comme séparateur.

Pour pouvoir écrire simplement dans le fichier il est nécessaire d'avoir un mot qui recherche le premier enregistrement libre. Ce mot choisira comme enregistrement libre le premier contenant un blanc comme premier caractère.

De même il faudra écrire un mot qui soit capable de retrouver l'enregistrement dont le champ spécifié contient une certaine information fournie par l'utilisateur.

Ces deux mots contiennent une partie commune, ce qui les différencie est le type de comparaison à effectuer. Nous utiliserons donc l'exécution vectorisée avec le mot RECHERCHE.

```
: RECHERCHE                              ( ––– )
        1 CM DEB @ DO                    (1 ––– )
                I ENRG !
                RECH1 @ EXECUTE          (1 Flag –––)
                IF 0= LEAVE THEN         (1 ––– )
        LOOP

; Return OK
```

Écrivons maintenant le mot LIBRE qui recherche le premier enregistrement vide et qui met son numéro dans ENRG :

```
: LIBRE1                       ( ––– Flag)
        ENRG–>ADR C @ 32 =     (Flag ––– )
; Return OK
: PLEIN                        ( ––– )
        ." FICHIER PLEIN"
; Return OK
: LIBRE                        ( ––– )
        ' LIBRE1 2 – RECH1 !
        ' PLEIN 2 – RECH2 !
        0 DEB ! RECHERCHE
; Return OK
```

Il est à noter que la recherche d'un enregistrement libre débute ici au numéro 0 (0 DEB !). Il est tout à fait possible de rajouter un mot qui permettra de ne pas balayer tout le fichier à chaque fois : de nombreux algorithmes existent. Pour des fichiers de petite taille le mot défini ci-dessus présente l'avantage d'utiliser des enregistrements effacés.

Le mot CHERCH qui recherche un enregistrement dont le champ spécifié correspond au texte voulu s'écrit :


```
: CHERCH1                      ( ––– Flag)
        PAD RECH @ ?CHAMP      (Adr Long –––)
        SWAP –TEXT 0=          (Flag –––)
; Return OK
: MANQUE                       ( ––– )
        ." INEXISTANT"
; Return OK
: CHERCH                       ( ––– )
        [COMPILE] ' 2+ RECH !
        ' CHERCH1 2 – RECH1 !
        ' MANQUE 2 – RECH2 !
        0 DEB ! 44 TEXT
        RECHERCHE
; Return OK
```

Le mot CHERCH s'utilise de la façon suivante :

CHERCH CHAMP XXXXXXXXXXXXX

où CHAMP est NOM ou PRENOM ou TELEPHONE ou SOCIETE
et XXXXXXXXXXXXXXX est le texte recherché.

Si votre FORTH ne dispose pas des mots TEXT et –TEXT voici leur définition :

```
: TEXT PAD 72 32 FILL WORD HERE COUNT PAD SWAP CMOVE ;
: –TEXT 2DUP + SWAP DO DROP 2+ DUP 2 – @ I @ – DUP IF DUP
ABS / LEAVE THEN 2 +LOOP SWAP DROP ;
```

Nous pouvons maintenant définir les mots dont se servira l'utilisateur :

```
: ECRIT
          LIBRE
          NOM CHAMP!
          PRENOM CHAMP!
          TELEPHONE CHAMP!
          SOCIETE CHAMP!
          FLUSH
; [Return] OK
```

Pour enregistrer par exemple Dupont Louis 9999999 Forth, il faudra taper :

```
ECRIT DUPONT,LOUIS,9999999,FORTH [Return] OK
```

```
: IMPRESSION
          NOM .CHAMP
          PRENOM .CHAMP
          TELEPHONE .CHAMP
          SOCIETE .CHAMP
; [Return] OK
```

Ce mot permet d'afficher le contenu de l'enregistrement pointé par ENRG.

```
: CHERCHE
          CHERCH IMPRESSION
; [Return] OK
```

Ce mot s'utilise de la même façon que CHERCH.

```
: AUTRE
          SUITE ENRG @  DEB !
          RECHERCHE IMPRESSION
; [Return] OK
```

Après avoir effectué une première recherche par l'intermédiaire du mot CHERCHE, ce mot permet de voir s'il existe dans la suite du fichier un

autre enregistrement qui concorde. Son utilisation peut-être répétée jusqu'à ce que le fichier ait été entièrement parcouru. Il est d'ailleurs très simple d'écrire un mot qui fasse une recherche de tous les correspondants quelque-soit leur nombre en utilisant CHERCHE puis AUTRE dans une structure BEGIN...WHILE...REPEAT.

```
: CHANGE
          [COMPILE]
          CHAMP!
; [Return] OK
```

Ce mot permet, une fois positionné sur le bon enregistrement, de modifier le contenu de l'un des champs de la façon suivante :

```
CHANGE CHAMP XXXXXX [Return] OK
```

où CHAMP est NOM ou PRENOM ou TELEPHONE ou SOCIETE
et XXXXXX est le nouveau texte.

```
: SUPPRIME
          ENRG->ADR LE 32 FILL
          UPDATE FLUSH
; [Return] OK
```

Ce mot permet, une fois positionné sur un enregistrement, de le supprimer en tapant :

```
SUPPRIME [Return] OK
```

Il est bien sûr possible de modifier les valeurs des différentes constantes choisies dans cet exemple. Les valeurs que nous avons choisies présentent l'avantage de fournir un listing directement exploitable simplement en tapant LIST des blocs qui constituent le fichier.

D'autres mots utilisateur peuvent être définis simplement à partir de ceux que nous avons donnés, cela dépend principalement du type d'application envisagée.

Le problème consiste à calculer la transformée de Fourier discrète d'un signal discret x(n). La formule de base est la suivante :

```
            N-1
X(k)=SOMME [x(n)*exp(-2*pi*j*k*n/N)]
            n=0
```

Ceci pour k=0,...,N−1

Nous allons calculer cette transformation à l'aide de l'algorithme Papillon. Ce type d'algorithme est plus couramment appelé F.F.T. (Fast Fourier Transform).

Le principe de cet algorithme est de décomposer la somme à calculer en deux sommes, la première regroupant les termes d'indices pairs et la seconde les termes d'indices impairs :

```
        (N/2)-1
X(k)= SOMME [x(2*i)*exp(-4*pi*j*i*k/N)]
          i=0

                                  (N/2)-1
  + exp(-pi*j*k/N) * SOMME [x((2*i)+1)*exp(-4*pi*j*i*k/N)]
                                    i=0
```

Le simple fait de réécrire cette formule permet de diminuer le nombre d'opérations à effectuer :

Pour N=8 nous avons 80 multiplications au lieu de 128 et 64 additions au lieu de 112.

En répétant cette opération une autre fois, il reste pour N=8 64 multiplications et 48 additions.

Cette dichotomie sera donc répétée jusqu'à ce que l'on arrive à des termes en somme de deux éléments [cf note].

Le but de ce type d'algorithme est bien sûr de limiter le temps de calcul. Considérant que plusieurs calculs de F.F.T. sont généralement effectués à la suite nous allons remplacer le réarrangement des éléments par une indirection. Ceci veut dire que nous allons construire un vecteur contenant les indices des éléments (x(n)) et que nous allons, lors d'une étape d'initialisation, calculer les nouveaux indices.

Voir *Le filtrage numérique* Walid P. SALMAN — Marc S. SOLOTAREFF. Éditions EYROLLES 1982.

Exemple : N=8

Indice initial	Indice après réarangement
0	0
1	4
2	2
3	6
4	1
5	5
6	3
7	7

Il est important de noter que le réarrangement des indices revient à faire du binaire réfléchi.

Exemple :

011 (=3) devient 110 (=6)

Dans la suite nous considérerons que N est une puissance de 2 : N=2puissM. Nous utiliserons donc une variable qui contiendra la valeur de M et lorsque nécessaire calculerons N à l'aide du mot 2PUISS.

```
M VARIABLE M Return OK

: 2PUISS                    (N --- 2puissN)
      DUP IF                (N --- )
          1 SWAP 0 DO
                  2 *
                LOOP
        ELSE
           DROP 1
        THEN
; Return OK
```

Définissons un mot de définition d'un vecteur dont la dimension (en nombre d'octets) est placée sur la pile.

```
: DEFVEC1                   (n --- )
      <BUILDS
           ALLOT
      DOES>
           +
; Return OK
```

Définissons maintenant le vecteur d'indices INDICE :

```
N DEFVEC1 INDICE Return OK
```

L'étape qui suit est de définir un mot REFL qui calcule le binaire réfléchi d'un nombre.


```
:    .                          (N --- Nb-bits --- N')
   0 SWAP                       (N 0 Nb-bits ---)
   0 DO                         (N 0 ---)
       OVER I 2PUISS AND        (N Flag ---)
       IF
         R> I SWAP >R
         I - 1 - 2PUISS OR
       THEN
   LOOP
   SWAP DROP
; Return OK
```

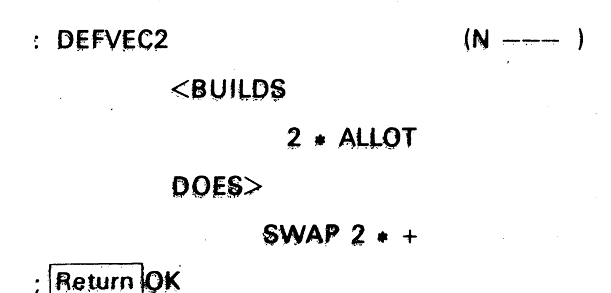
Nous pouvons maintenant définir le mot INIT–INDICE qui va écrire dans le tableau d'indirection INDICE les binaires réfléchis des indices de départ.

```
: INIT-INDICE
     M @ 2PUISS 0 DO
                 I DUP REFL SWAP
                 INDICE C!
              LOOP
; Return OK
```

Nous allons maintenant nous intéresser aux vecteurs qui vont représenter la transformée de Fourier discrète.

Les résultats sont complexes en raison des exponentielles complexes qui apparaissent dans la formule. Nous décomposerons ces exponentielles et X(k) en partie réelle et partie imaginaire.

Nous allons donc définir deux vecteurs de dimension N. Pour ce faire utilisons le mot de définition suivant :



```
: DEFVEC2                   (N --- )
      <BUILDS
           2 * ALLOT
      DOES>
           SWAP 2 * +
; Return OK
```

Nous construisons alors XR (partie réelle) et XI (partie imaginaire) de la façon suivante:

```
M @  2PUISS DEFVEC2 XR Return OK
M @  2PUISS DEFVEC2 XI Return OK
```

Pour pouvoir écrire et lire les éléments de ces vecteurs nous définissons les mots suivants:

```
: CSTORE                    (R Im Indice ———)
          SWAP   OVER       (R Indice Im Indice ———)
          XI !              (R Indice ———)
          XR !              (——— )
; Return OK

: CGET                      (Indice ——— R Im)
       INDICE @ DUP         (I' I' ———)
       XR @                 (I' R ———)
       SWAP                 (R I' ———)
       XI @                 (R Im ———)
; Return OK
```

Ayant à utiliser un pas de boucle variable nous définirons:

```
VARIABLE LE          VARIABLE LE1
```

Nous aurons également besoin de deux variables complexes U et W:

```
1 0 COMPLEX U
1 0 COMPLEX W
```

La lecture et l'écriture de ces variables se feront par l'intermédiaire des mots X @ et X! définis dans le problème 1.

Nous supposerons que le signal x(n) dont nous cherchons à calculer la transformée de Fourier discrète a ses valeurs dans le vecteur XR.


```
: FFT1
     M @ 2PUISS 1+ SWAP DO
            I LE1 @  + DUP >R
            CGET U X  @ C*
            I CGET 2OVER C— R> CSTORE
            I CGET C+ I CSTORE
     LE @  +LOOP
; Return OK

: FFT
     M @  1+ 1 DO
            I 2PUISS DUP LE !
            2 / LE1 !
            1 0 U X!
            180 LE1 @  / MINUS DUP
            COS SWAP SIN W X!
            LE1    1+ 1 DO
                   I FFT1
                   U X @ W X @ C*
                   U X!
                   LOOP
     LOOP
; Return OK
```

ANNEXE I

LISTE DES MOTS COMPOSANT LE FORTH-79

!	(n adr ---)	Écrit n à l'adresse adr "Store"
#	(ud1 --- ud2)	Génère à partir d'un nombre non-signé double longueur un caractère ASCII qui est placé dans une chaîne de sortie. Le résultat ud2 est le quotient de la division de ud2 par BASE. Ce mot est situé entre <# et #>. "Sharp"
#>	(d --- adr n)	Termine le formattage pour une édition. Supprime d et place sur la pile l'adresse du texte à éditer, ainsi que le nombre de caractères dans l'ordre voulu pour TYPE. "Sharp-greater"
#S	(ud --- 0 0)	Convertit chaque chiffre d'un nombre double longueur (32 bits) non-signé en un caractère qui est intégré dans le texte à éditer. Si le nombre est nul, un seul zéro est écrit. Ce mot n'est utilisable qu'entre les mots <# et #>. "Sharp-S"
'	(--- adr) ' MOT	Lorsque l'interpréteur est en mode exécution, ce mot place sur la pile le Pfa du mot qui suit dans l'input stream (MOT). Lorsque l'interpréteur est en mode compilation, ce mot charge à la suite du dictionnaire le Pfa du mot qui suit dans l'input stream. L'interpréteur considère en fait ce Pfa comme un nombre (LITERAL) situé dans la définition. "Tick"

(	(---)	Ce mot indique que le texte qui suit est du commentaire, délimité par :) . C'est un mot qui doit donc être encadré par deux séparateurs. Ce mot peut-être utilisé de la même façon que l'interpréteur soit en mode exécution ou compilation. " Paren"
*	(n1 n2 --- n3)	Effectue la multiplication de n1 par n2 et laisse le résultat n3 sur la pile. "Times"
*/	(n1 n2 n3 --- n4)	Multiplication n1 par n2, divise le résultat par n3 et laisse le résultat final n4 sur la pile. Le résultat intermédiaire (n1*n2) est en double longueur (32 bits), ce qui permet d'obtenir une plus grande précision que la séquence : n1 n2 * n3 / "Times-divide"
*/MOD	(n1 n2 n3 --- n4 n5)	Multiplie n1 par n2, divise le résultat en double longueur par n3 et place le reste n4 de la division puis le quotient n5 sur la pile. "Times-divide-mod"
+	(n1 n2 --- n3)	Effectue l'addition de n1 et n2 puis place le résultat n3 sur la pile. "Plus"
+!	(n adr ---)	Ajoute n à la valeur sur 16 bits située à l'adresse adr et mémorise le résultat à cette même adresse. " Plus-store"
+LOOP	(n ---)	Ajoute n à l'indice de boucle et compare ce nouvel indice à la limite de la boucle. Renvoie l'exécution au DO correspondant tant que l'indice calculé est strictement inférieur à la limite. Sinon termine la boucle en éliminant les paramètres sur la Return Stack et en poursuivant l'exécution. "Plus-loop"
,	(n ---)	Réserve deux octets à la suite du dictionnaire et y place n. "Comma"
−	(n1 n2 --- n3)	Soustrait n2 de n1 et laisse le résultat n3 sur la pile. " Minus"

−TRAILING	(adr n1 --- adr n2)	Examine dans une chaîne de n1 caractères commençant en mémoire à l'adresse adr, à partir de quel caractère la chaîne ne contient plus que des blancs. Les caractères entre l'adresse adr+n2 et l'adresse adr+n1−1 sont tous des blancs. "Dash−trailing"
.	(n ---)	Convertit le nombre n dans la base courante (BASE) et affiche la valeur, éventuellement avec un signe moins, ainsi qu'un blanc de séparation à la suite. "Dot"
."	."texte"	Affiche le texte qui suit ce mot. La fin de texte est signalée par le caractère ". La longueur du texte doit être inférieure à la dimension du buffer de sortie. "Dot−quote"
/	(n1 n2 --- n3)	Divise n1 par n2 et laisse le quotient n3 sur la pile : c'est une division entière. "Divide"
/MOD	(n1 n2 --- n3 n4)	Divise n1 par n2 et laisse le reste n3 puis le quotient n4 sur la pile. "Divide−mod"
0<	(n --- flag)	Compare n à zéro et laisse un flag sur la pile : flag=1 si n<0 flag=0 sinon "zéro-less"
0=	(n --- flag)	Compare n à zéro et laisse un flag sur la pile : flag=1 si n=0 flag=0 sinon "zéro−equal"
0>	(n --- flag)	Compare n à zéro et laisse un flag sur la pile : flag=1 si n>0 flag=0 sinon "zéro−greater"
1+	(n --- n+1)	Incrémente de un le sommet de la pile. "One−plus"
1−	(n --- n−1)	Décrémente de un le sommet de la pile. "One−minus"

2+	(n --- n+2)	Incrémente de deux le sommet de la pile. "Two-plus"
2–	(n --- n–2)	Décrémente de deux le sommet de la pile. "Two-minus"
:	: MOT définition	Le vocabulaire courant (CURRENT) devient aussi le vocabulaire de contexte (CONTEXT). Une tête de chaîne pour MOT est alors créée dans ce vocabulaire, et l'interpréteur passe en mode compilation. Les mots qui suivent dans l'input stream et qui sont immédiats sont exécutés, le Cfa des autres est placé à la suite du dictionnaire. Si l'un de ces mots n'est pas trouvé par l'interpréteur dans les vocabulaires de contexte et FORTH, celui-ci teste si le nom peut correspondre à un nombre dans la base courante : si oui il charge cette valeur comme un nombre à la suite du dictionnaire. Sinon un message d'erreur est affiché et la compilation est interrompue. "Colon"
;		Indique la fin de définition d'un mot. L'interpréteur repasse en mode exécution. "Semi-colon"
<	(n1 n2 --- flag)	Compare n1 à n2 et laisse un flag sur la pile : flag=1 si n1<n2 flag=0 sinon "Less–than"
<#		Initialise le formattage pour une édition. La séquence : <# # #S HOLD SIGN #> permet d'afficher un nombre en double longueur. "Less–sharp"
=	(n1 n2 --- flag)	Compare n1 à n2 et laisse un flag sur la pile : flag=1 si n1=n2 flag=0 sinon "Equals"

>	(n1 n2 --- flag)	Compare n1 à n2 et laisse un flag sur la pile : flag=1 si n1>n2 flag=0 sinon "Greater–than"
>IN	(--- adr)	Laisse sur la pile l'adresse d'une variable qui contient l'offset du caractère suivant dans l'input stream. "To–in"
>R	(n ---)	Transfère le sommet de la pile de données vers la pile de Return. "To–r"
?	(adr ---)	Affiche le contenu de la mémoire à l'adresse adr. "Question–mark"
?DUP	(n --- n n) ou (n --- n)	Duplique le sommet de la pile si celui-ci est non nul. "Query–dup"
@	(adr --- n)	Empile le contenu n de la mémoire à l'adresse adr. "Fetch"
ABORT		Nettoie les deux piles, met l'interpréteur en mode exécution et rend la main à l'utilisateur. "Abort"
ABS	(n1 --- n2)	Calcule la valeur absolue n2 du sommet de la pile n1 et l'empile. "Absolute"
ALLOT	(n ---)	Ajoute n mots au Parameter Field du dernier mot créé. "Allot"
AND	(n1 n2 --- n3)	Effectue un ET logique bit à bit entre les deux nombres.
BASE	(--- Adr)	Empile l'adresse d'une variable contenant la base courante de conversion des entrées-sorties.
BEGIN	BEGIN...flag WHILE...REPEAT BEGIN...flag UNTIL BEGIN...AGAIN	BEGIN marque le début d'une séquence répétitive de mots. Une boucle BEGIN–UNTIL s'exécutera tant que le flag est faux. Une boucle BEGIN–WHILE–REPEAT s'exécutera tant que le flag est vrai.

BLK	(––– Adr)	Empile l'adresse d'une variable contenant le numéro du block d'entrée-sortie étant interprété comme input stream. Si le contenu est zéro, l'input stream est le terminal.
BLOCK	(n ––– Adr)	Empile l'adresse du premier octet du bloc n. Si le bloc n'est pas résident, il est transféré depuis la mémoire de masse dans le buffer le moins récemment accédé. Si le bloc occupant ce buffer a été modifié (UPDATE), il est recopié sur la mémoire de masse. Si les accès à la mémoire de masse échouent, c'est une condition d'erreur. Seule l'adresse du dernier bloc référencé par BLOCK est valide, à cause du partage des buffers.
BUFFER	(n ––– Adr)	Réserve le prochain buffer, en lui assignant le numéro de Bloc n. Le bloc n'est pas lu sur la mémoire de masse. Si le précédent contenu du buffer est un bloc modifié (UPDATE), il est sauvegardé dans la mémoire de masse. Si les accès échouent, c'est une condition d'erreur. L'adresse empilée est celle du premier octet du buffer.
C!	(n Adr –––)	Mémorise l'octet le moins significatif de n à l'adresse Adr. "C–store"
C@	(Adr ––– Octet)	Empile le contenu de l'octet d'adresse Adr (Les huits bits de poids fort sont à zéro) "C–fetch"
CMOVE	(Ad1 Ad2 n –––)	Déplace n octets de l'adresse Ad1 vers l'adresse Ad2. Le transfert commence à l'octet d'adresse Ad1. Aucun effet si n est négatif ou nul.
COMPILE		Lorsqu'un mot contenant COMPILE est exécuté, la valeur sur 16 bits suivant le Pfa de COMPILE est compilée (placée dans le dictionnaire). Par exemple, COMPILE DUP compilera le Cfa de DUP. COMPILE [0 ,] compilera 0.
CONSTANT	n CONSTANT <nom>	Créé une définition de <nom> et place n dans son Pfa. Lorsque <nom> sera exécuté, il empilera n.

CONTEXT	(––– Adr)	Empile l'adresse d'un variable spécifiant le vocabulaire dans lequel seront faites les recherches pendant l'interprétation de l'input stream.
CONVERT	(d1 ad1 ––– d2 ad2)	Convertit dans une cellule double de la pile (d1, d2) la chaîne de caractère commençant à l'adresse ad1+1. L'adresse Ad2 représente celle du premier caractère non convertible dans la base courante.
COUNT	(Ad ––– Ad+1 n)	Empile l'adresse Ad+1 et le nombre de caractères du texte commençant à Ad. Le premier octet à l'adresse Ad doit contenir la longueur du texte.
CR		Émet un retour chariot sur le périphérique de sortie courant.
CREATE	CREATE <nom>	Crée une définition dans le dictionnaire pour <nom>, sans réserver de Pfa. Lorsque <nom> sera exécuté, son Pfa sera empilé.
CURRENT	(––– Ad)	Empile l'adresse d'une variable spécifiant le vocabulaire dans lequel les nouvelles définitions doivent être placées.
D+	(d1 d2 ––– d3)	Additionne d1 et d2. "D–plus"
D<	(d1 d2 ––– flag)	Vrai si d1 est inférieur à d2. "D–inférieur"
DECIMAL		Fait passer en base 10 pour les entrées-sortie.
DEFINITIONS		Rend le vocabulaire de contexte vocabulaire courant, ainsi toutes les nouvelles définitions seront créées dans le vocabulaire de contexte.
DEPTH	(––– n)	Empile le nombre de cellules de 16 bits présentes sur la pile de données (n excepté).
DNEGATE	(d ––– –d)	Empile le complément à deux d'un nombre en double longueur.

DO	(n1 n2 ---) DO ... LOOP DO ... +LOOP	Début d'une boucle qui se terminera suivant la valeur de paramètres de contrôle. L'indice de boucle commence à n2, et se termine selon la limite n1. A LOOP ou +LOOP, l'indice est modifié d'une valeur positive ou négative. Les boucles DO...LOOP doivent pouvoir être imbriquées à au moins trois niveaux.
DOES>	: <nom> ... CREATE ... DOES> ... ; <nom> <nomex>	Définit la partie exécution d'un mot créé par un mot de haut niveau. Marque la fin de la partie de définition du mot de définition <nom>, et commence la définition de la partie exécution des mots définis à l'aide de <nom>. A l'exécution de <nomex>, la séquence de mots entre DOES> et ; sera exécutée, le Pfa de <nomex> étant sur la pile.
DROP	(n ---)	Jette le nombre en simple longueur situé au sommet de la pile. "Drop"
DUP	(n --- n n)	Duplique le nombre en simple longueur situé au sommet de la pile. "Dup"
ELSE	IF...ELSE...THEN	Lorsque ELSE est utilisé dans cette structure à l'intérieur d'une définition il s'exécute après la partie VRAI suivant IF. Il déroute l'exécution juste après le THEN. Il n'a pas d'effet sur la pile. "Else"
EMIT	(Carac. ---)	Transmet le caractère au périphérique de sortie courant. "Emit"
EMPTY-BUFFERS		Met à VIDE les flags d'occupation de tous les buffers d'entrée-sortie, sans nécessairement les mettre à zéro. Les buffers qui ont été modifiés ne sont pas sauvegardés. "Empty-buffers"
EXECUTE	(Adr ---)	Exécute le mot du dictionnaire dont le Cfa est sur la pile. "Execute"
EXIT		Lorsque utilisé dans une définition, stoppe l'exécution à cet endroit. Ne doit pas être utilisé à l'intérieur d'une structure DO ... LOOP. "Exit"

EXPECT	(Adr n ---)	Transfère les caractères venant du périphérique d'entrée dans la zone mémoire commençant à l'adresse Adr, jusqu'à réception de Return, ou celle de n caractères. N'a aucun effet si n est inférieur ou égal à 0. Un ou deux nuls sont placés à la fin du texte en mémoire. "Expect"
FILL	(Adr n Octet ---)	Remplit la zone mémoire commençant à l'adresse Adr de n octets de valeur Octet. Aucune action si n est inférieur ou égal à 0. "Fill"
FIND	(--- Adr)	Laisse sur la pile le Cfa du mot suivant de l'Input Stream. Si ce mot n'est pas trouvé après une recherche dans le vocabulaire de contexte et dans le vocabulaire FORTH, laisse 0 sur la pile. "Find"
FORGET	FORGET <nom>	Supprime du dictionnaire <nom>, du vocabulaire courant, et tous les mots définis depuis, sans distinction de vocabulaire. Si la recherche n'aboutit pas dans le vocabulaire courant et dans FORTH, un message d'erreur est émis. "Forget"
FORTH		Le nom du vocabulaire de base. Les nouvelles définitions lui sont reliées, jusqu'à assignation d'un nouveau vocabulaire courant. Les vocabulaire courant d'un utilisateur lui sont chaînés, et on peut donc considérer que tous les vocabulaires contiennent FORTH. "Forth"
HERE	(--- Adr)	Retourne l'adresse de la première cellule libre du dictionnaire. "Here"
HOLD	(Char ---)	Insère un caractère dans une chaîne représentant un nom-formaté pour l'édition. Doit être utilisé uniquement dans la structure <#...#>. "Hold"
I	(--- n)	Copie l'indice de boucle sur la pile de donnée. Fait partie des structures DO ... I ... LOOP et DO ... I ... +LOOP. "I"

IF	(Flag ---)	Utilisé dans les syntaxes IF...ELSE...-THEN et IF...THEN. Si le Flag est VRAI, les mots suivant IF sont exécutés et les mots suivant ELSE ne le sont pas. Cette seconde partie de la structure n'est pas indispensable. Si le Flag est FAUX, les mots entre IF et ELSE, ou ceux entre IF et THEN (si il n'y a pas de ELSE), ne sont pas exécutés. Ces structures peuvent être imbriquées. "If"
IMMEDIATE		Marque le dernier mot défini dans le dictionnaire comme un mot à exécuter et non pas à compiler, lorsqu'on le rencontre pendant une compilation. "Immediate"
J	(--- n)	Retourne l'indice de la boucle de niveau immédiatement supérieur, dans les structures de boucles imbriquées: DO...DO...J...LOOP...LOOP. "J"
KEY	(--- Carac.)	Laisse sur la pile la valeur ASCII du prochain caractère arrivant du périphérique d'entrée. "Key"
LEAVE		Force la sortie d'une boucle DO...-LOOP au prochain LOOP ou +LOOP en mettant l'indice final de la boucle égal à la valeur de l'indice courant. L'indice lui-même reste inchangé, et l'exécution se déroule normalement jusqu'à la rencontre du mot de fin de boucle. "Leave"
LIST	(n ---)	Liste le contenu de l'écran n sur le périphérique de sortie, et met n dans la user-variable SCR. "List"
LITERAL	(n ---)	En cours de compilation, compile la valeur n comme un litéral sur 16 bits, qui lorsqu'il sera exécuté, laissera la valeur n sur la pile de donnée. "Literal"

LOAD	(n ---)	Lance l'interprétation de l'écran n en le prenant comme Input Stream; préserve les pointeurs de l'actuel Input Stream (de >IN à BLK). Si l'interprétation ne se termine pas explicitement, elle se terminera à la fin de l'écran. Le contrôle est alors passé à l'Input Stream contenant le LOAD, déterminé par les pointeurs d'Imput Stream >IN et BLK.
LOOP		Incrémente de 1 l'indice de la boucle DO...LOOP, et termine la boucle si le nouvel indice est supérieur ou égal à la limite. "Loop"
MAX	(n1 n2 --- n3)	Laisse le plus grand des deux nombres sur la pile. "Max"
MIN	(n1 n2 --- n3)	Laisse le plus petit des deux nombres sur la pile. "Min"
MOD	(n1 n2 --- n3)	Divise n1 par n2, et laisse le reste, avec le même signe que n1. "Mod"
MOVE	(Adr1 Adr2 n ---)	Déplace un nombre n de mots de 16 bits, de l'adresse Adr1 vers l'adresse Adr2. Le premier mot transféré est le mot d'adresse Adr1. Si n est négatif ou nul, aucun effet. "Move"
NEGATE	(n --- -n)	Laisse sur la pile le complément à 2 du nombre, c'est-à-dire la différente 0-n. "Negate"
NOT	(Flag1 --- Flag2)	Inverse le sens booléen du flag sur la pile. Identique au mot 0=. "Not"
OR	(n1 n2 --- n3)	Laisse le résultat du OU inclusif bit par bit des deux nombres.
OVER	(n1 n2 --- n1 n2 n1)	Empile la copie du second nombre.
PAD	(--- Adr)	Empile l'adresse d'une zone tampon utilisée pour ranger les chaines de caractère à interpréter. La capacité minimun de PAD est 64 caractères.

PICK	(n1 ––– n2)	Empile le contenu de la n1 ième cellule de la pile. Si n1 est inférieur à 1, un message d'erreur est émis. 2 PICK est équivalent à OVER.
QUERY		Accepte 80 caractères venant du périphérique d'entrée, et les range dans le Terminal Input Buffer (TIB). WORD peut être utilisé pour utiliser ce buffer comme input stream, en mettant >IN et BLK à zéro.
QUIT		Initialise la pile de Return fait passer en mode interprétation, et passe le contrôle au terminal. Aucun message n'apparaît.
R>	(––– n)	Transfère n de la pile des Return à la pile des données. "R–from"
R@	(––– n)	Copie sur la pile de donnée le sommet de la pile des Return. "R–fetch"
REPEAT	BEGIN...WHILE... REPEAT	A l'exécution, REPEAT déroute juste après le BEGIN correspondant.
ROLL	(n –––)	Fait effectuer une permutation circulaire aux n premières cellules de la pile. 3 ROLL est équivalent à ROT.
ROT	n1 n2 n3 ––– n2 n3 n1	Fait effectuer une permutation circulaire aux trois premières cellules de la pile.
SAVE–BUFFERS		Sauvegarde tous les buffers d'entrée-sortie qui ont été modifiés. Si l'écriture sur mémoire de masse échoue, un message d'erreur est édité.
SCR	(––– Adr)	Empile l'adresse d'une variable qui contient le numéro du dernier écran listé. "S–c–r"
SIGN	(n –––)	Insère dans la chaîne de caractères en construction représentant un nombre le caractère ASCII – si n est négatif.
SPACE		Émet le caractère blanc ASCII vers le périphérique de sortie.
SPACES	(n –––)	Émet n caractères blanc ASCII vers le périphérique de sortie. Aucun effet si n est négatif ou nul.
SWAP	(n1 n2 ––– n2 n1)	Échange les deux cellules au sommet de la pile.
THEN	IF...ELSE...THEN IF...THEN	THEN est le point de déroutement de l'exécution après le ELSE, ou le IF s'il n'y a pas de ELSE.
TYPE	(Adr n –––)	Émet une zone mémoire de n octets commençant à l'adresse Adr vers le périphérique de sortie. Aucun effet si n est négatif ou nul.
U*	(Un1 Un2 ––– Ud3)	Effectue une multiplication non-signée de Un1 par Un2, et laisse le produit en double longueur sur la pile. Toutes valeurs non-signées. "U–fois"
U.	(Un –––)	Affiche la valeur de Un dans la base courante, en format libre, non-signé, suivi d'un blanc. "U–point"
U/MOD	(Ud1 Un2–––Un3 Un4)	Effectue la division non-signée du nombre en double longueur Ud1 par Un2, et retourne le reste Un3, et le quotient Un4. Toutes valeurs non-signées. "u–slash-mod"
U<	(Un1 Un2 ––– Flag)	Laisse le flag correspondant à Un1<Un2, ces deux valeurs étant considérées comme des entiers non-signés sur 16 bits. "U–inf"
UNTIL	(Flag –––)	Dans une définition, marque la fin d'une boucle BEGIN...UNTIL, qui se terminera selon la valeur du flag. S'il est vrai, la boucle est terminée. Si le flag est faux, l'exécution est déroutée vers le premier mot suivant BEGIN. Ces structures peuvent être imbriquées.
VARIABLE	VARIABLE <nom>	Créé une défition du dictionnaire pour <nom>, et réserve deux octets pour le stockage du Pfa. On doit initialiser la valeur. Quand on exécute <nom>, il empile l'adresse de stockage.

VOCABU-LARY	VOCABULARY <nom>	Crée, dans le vocabulaire courant, une définition de <nom> qui spécifie une nouvelle liste ordonnée de définitions de mots. Les exécutions ultérieures de <nom> en feront le vocabulaire de contexte. Quand <nom> sera vocabulaire courant (voir DEFINITIONS), les nouvelles définitions lui seront rattachées. Les nouveaux vocabulaires sont chaînés à FORTH.
WHILE	(Flag ---) BEGIN... flag WHILE...REPEAT	Sélectionne l'exécution conditionnelle suivant la valeur de flag. S'il est vrai, continue l'exécution jusqu'au REPEAT, qui la déroutera au premier mot après BEGIN. Si le flag est faux, l'exécution est déroutée juste après le REPEAT.
WORD	(Carac --- Adr)	Reçoit des caractères de l'input stream jusqu'à un caractère délimiteur, ou la fin de l'input stream. La chaîne de caractères est stockée, précédée de sa longueur. Le caractère de fin Carac ou nul est placé à la fin de la chaîne, et ne compte pas dans sa longueur. Si l'input Stream est terminé quana WORD est exécuté, la longueur est nulle. L'adresse du premier octet de la chaîne est empilée.
XOR	(n1 n2 --- n3)	Effectue un OU exclusif bit par bit entre les deux nombres.
[		Sort du mode compilation. Le texte qui suit est exécuté. Voir]. "Crochet-gauche"
[COMPILE]	[COMPILE] <nom>	Force la compilation du mot suivant. Permet de compiler un mot immédiat, qui aurait été exécuté.

Extensions

2!	(d adr ---)	Range en mémoire la valeur d sur quatre octets à partir de l'adresse adr. (double longueur). "two–store"
2@	(adr --- d)	Ramène sur la pile le contenu des quatres cellules mémoire à partir de l'adresse adr. (double longueur). "two–fetch"

2CONSTANT	(d ---) d 2CONSTANT <name>	Crée dans le dictionnaire une tête de chaîne et range la valeur d dans le parameter field, en double longueur. "two–constant"
2DROP	(d ---)	Dépile le sommet de la pile en double longueur. "two–drop"
2DUP	(d --- d d)	Duplique le sommet de la pile en double longueur. "two-dup"
2OVER	(d1 d2 --- d1 d2 d1)	Créé une copie du nombre en seconde position dans la pile, en double longueur. "two-over"
2ROT	(d1 d2 d3 --- d2 d3 d1)	Permutation circulaire gauche des trois éléments en double longueur situés au sommet de la pile. "two-rot"
2SWAP	(d1 d2 --- d2 d1)	Échange les deux éléments au sommet de la pile en double longueur "two–swap"
2VARIABLE	(d ---) 2VARIABLE <name>	Crée une tête de chaîne pour une variable en double longueur. Réserve quatre octets pour la valeur. "two-variable"
D+	(d1 d2 --- d1+d2)	Addition en double longueur. "d–plus"
D–	(d1 d2 --- d1–d2)	Soustraction en double longueur. "d–minus"
D.	(d ---)	Affiche un nombre en double longueur. "d–dot"
D.R	(d n ---)	Affiche un nombre, justifié à droite, avec signe si négatif. "d–dot–r"
D0=	(d --- flag)	Compare une valeur numérique en double longueur à zéro et retourne un flag. "d–zéro–équals"
D<	(d1 d2 --- f)	Compare deux nombres en double longueur. "d–less"

D=	(d1 d2 ––– f)	Compare deux nombres en double longueur. "d–equal"
DABS	(d1 ––– d2)	Retourne la valeur absolue d'un nombre en double longueur. "d–abs"
DMAX	(d1 d2 ––– d3)	Retourne sur la pile le plus grand de deux nombres. "d–max"
DMIN	(d1 d2 ––– d3)	Retourne sur la pile le plus petit de deux nombres. "d–min"
DNEGATE	(d ––– –d)	Retourne sur la pile le complément à deux d'un nombre en double longueur. "d–negate"
DU<	(ud1 ud2 ––– f)	Même fonction que D<, avec des nombres en double longueur non signés. "d–u–less"

Assembleur

;CODE	syntaxe : : <name>... ;CODE	Arrête la compilation et termine le mot de définition <name>. ASSEMBLER devient le vocabulaire de contexte. Lorsque <name> est exécuté sous la forme : <name> <namex> pour définir <namex>, l'adresse d'exécution de <namex> contiendra l'adresse de la séquence de mode contenue dans la définition de <name>, qui suit ,CODE. L'exécutio de <namex> entrainera l'exécution de la séquence de code correspondante. "semi–colon–code"
ASSEMBLER		Sélectionne le vocabulaire ASSEMBLER comme vocabulaire de contexte.
CODE	syntaxe : CODE <name>... ... END–CODE	Mot de définition utilisé pour générer une tête de chaîne <name>, défini par la séquence de code incluse entre CODE et END–CODE. ASSEMBLER devient vocabulaire de contexte.
END–CODE		Fin de "code-définition". Remet CONTEXT à CURRENT, si aucune erreur n'a été détectée en cours de définition.

Candidats à la standardisation

!BITS	(n1 adr n2 –––)	Range en mémoire la valeur n1 masquée par n2, à l'adresse adr, elle aussi masquée par n2. "store–bits"
**	(n1 n2 ––– n3)	Fonction exponentiation. "power"
+BLOCK	(n1 ––– n2)	Retourne sur la pile n1 augmenté du nombre de blocs interprétés. "plus–block"
–'	(––– adr 0) ou (––– 1) syntaxe : –' <name>	Si <name> peut être trouvé dans le vocabulaire de contexte, retourne son PFA et un flag FALSE, sinon retourne un flag TRUE. "dash–tick"
–>		Poursuit l'interprétation dans le bloc suivant. Peut être utilisé dans une définition trop longue... "next–block"
–MATCH	(ad1 n1 ad2 n2 ––– ad3 f)	Comparaison du contenu de deux zones mémoire. Les n2 premiers caractères, à partir de l'adresse ad2 sont comparés aux n1 premiers caractères à partir de l'adresse ad1. Retourne sur la pile l'adresse du premier caractère en défaut (ad3) et un flag qui indique l'égalité des deux zones. "dash–match"
–TEXT	(ad1 n1 ad2 ––– ––– n2)	Compare deux chaînes, sur n1 caractères, à partir des adresses ad1 et ad2. Retourne la différence des deux premiers caractères différents, ou 0. "dash–text"
.R	(n1 n2 –––)	Affiche n1, justifié à droite, dans un champ de n2 caractères. si n2 est plus petit que 1, aucun effet. "dot–r"
/LOOP	(n –––)	Mot de fin de structure DO LOOP. L'indice de boucle est augmenté de la valeur non-signée de n. "up loop"
1+!	(adr –––)	Ajoute 1 à la valeur sur 16 bits à l'adresse adr. "one–plus–store"
1–!	(adr –––)	Retranche 1 à la valeur sur 16 bts situées à l'adresse adr. "one–minus–store"
2*	(n1 ––– n2)	Multiplication par 2. "two-times"

2/	(n1 --- n2)	Division par 2. "two-divide"
;S		Stoppe l'interprétation d'un bloc. Exécution uniquement. "semi–s"
<>	(n1 n2 --- flag)	Teste l'inégalité de deux nombres et retourne un flag au sommet de la pile. "not–equal"
<BUILDS	syntaxe : : <name> ... <BUILDS (1) DOES> (2) ; <name> <namex>	Outil de définition de mot de définition. Lorsque <name> sera exécuté, <BUILDS crée une tête de chaine pour <namex>, le parameter field est préparé par la séquence (1). A l'exécution de <namex>, la séquence (2) sera exécutée.
<CMOVE	(ad1 ad2 n ---)	Copie n octets à partir de l'adresse ad1 à l'adresse ad2, en commençant par les adresses les plus hautes. "reverse–c–move"
><	(n1 --- n2)	SWAP l'octet de poids fort avec l'octet de poids faible. "byte-swap"
>MOVE<	(ad1 ad2 n ---)	Copie n octets de ad1 vers ad2. L'ordre des octets est inversé à chaque mot. "byte–swap–move"
@ BITS	(adr n1 --- n2)	Retourne sur la pile le contenu de l'adresse adr, masqué par n1 "fetch–bits"
ABORT"	(flag ---)	Si le flag est TRUE, affiche à l'écran tout le texte qui suit jusqu'à la rencontre de " et enchaîne sur ABORT. "abort–quote"
AGAIN		Saut inconditionnel au début de la structure BEGIN–AGAIN.
ASCII	exécution (--- c) compilation (---)	Retourne le code ASCII du premier caractère non-blanc de l'input stream. A la compilation, compile ce caractère comme un litéral, qui sera retourné sur la pile à l'exécution.
ASHIFT	(n1 n2 --- n3)	Décalage de n1 de n2 bits. n2⩾0 : décalage à gauche remplacement par des zéro. n2⩽0 : décalage à droite, avec conservation du signe.
B/BUF	(--- 1024)	Constante qui contient le nombre de caractères par buffer.
BELL		Déclenche un beep sonore sur le terminal.

BL	(--- n)	Retourne sur la pile la valeur ASCII du caractère blanc. "b–l"
BLANKS	(adr n ---)	Remplit une zone mémoire commençant à l'adresse adr de n caractères blancs.
C,	(n ---)	Range la valeur sur 8 bits au sommet de la pile à la première cellule libre du dictionnaire. Met à jour DP.
CHAIN	syntaxe ; CHAIN <name>	Connecte le vocabulaire CURRENT à toutes les définitions qui pourront être faites sous le vocabulaire <name> à l'avenir. Chaque vocabulaire ne peut être connecté qu'une fois, mais peut être l'objet de plusieurs chaînages. Par exemple, chaque vocabulaire peut contenir la séquence CHAIN FORTH.
COM	(n1 --- n2)	Retourne le complément à 1 de n1.
CONTINUED	(n ---)	Continue l'interprétation du block n.
CUR	(--- adr)	Variable qui retourne la position courante de la tête de lecture.
DBLOCK	(d --- adr)	Identique à BLOCK mais utilise un nombre non-signé en double longueur. "d–block"
DPL	(--- adr)	Variable qui contient la position de la virgule pour l'affichage. Si DPL vaut 0, un point décimal sera tout de même émis en dernière position. Aucun point ne sera généré si DPL est négatif. "d–p–l"
DUMP	(adr n ---)	Liste le contenu des n octets à partir de l'adresse adr.
EDITOR		EDITOR devient vocabulaire de CONTEXT.
END		Synonyme de UNTIL.
ERASE	(adr n ---)	Met à zéro n octets à partir de l'adresse adr.
FLD	(--- adr)	Variable qui pointe sur le champ réservé pour un nombre pendant les conversions pour affichage.
FLUSH		Un synonyme de SAVE–BUFFERS.
H.	(n ---)	Affichage d'un nombre en hexadécimal. La base courante est conservée. "h–dot"
HEX		Fixe la valeur de la base courante à seize.

I'	(--- n)	Utilisé pendant les définitions de mots, en particulier dans les structures DO—LOOP. Retourne sur la pile l'indice maximum de boucle, c'est-à-dire la deuxième valeur dans la pile des returns. "i—prime"
IFEND		Termine une structure conditionnelle 'IFTRUE'.
IFTRUE	(flag ---) syntaxe : IFTRUE ... OTHERWISE ... IFEND	Structure semblable à IF ELSE THEN, sauf qu'elle ne peut être imbriquée, et ne peut être utilisée qu'à l'interprétation. Permet de contrôler la compilation, sans être incluse dans la définition.
INDEX	(n1 n2 ---)	Affiche la première ligne de chaque écran de n1 à n2, qui contient conventionnellement le titre.
INTERPRET		Commence l'interprétation à partir du caractère indexé par le contenu de >IN, relativement au bloc adressé par BLK, jusqu'à ce que l'INPUT STREAM soit épuisé. Si BLK contient zéro, l'interprétation a lieu sur le TERMINAL INPUT BUFFER.
K	(--- n)	Retourne l'indice de la boucle externe pour deux structures DO LOOP imbriquées.
LAST	(--- adr)	Variable qui contient l'adresse de début de tête de chaîne de la dernière entrée du dictionnaire, qui n'est pas nécessairement une définition valide ou complète.
LINE	(n --- adr)	Retourne l'adresse du début de la ligne n du bloc adressé par SCR. Le numéro de ligne varie de 0 à 15.
LINELOAD	(n1 n2 ---)	Commence l'interprétation à partir de la ligne n1 de l'écran n2.
LOADS	(n ---) syntaxe : n LOADS <name>	Lorsque <name> sera exécuté, l'écran n sera chargé.
MASK	(n1 --- n2)	Retourne un masque avec les n1 bits de poids fort à 1 si n1>0, ou bits de poids faible, si n1<0.
MS	(n ---)	Introduit un délai d'exécution de environ n millisecondes.
NAND	(n1 n2 --- n3)	Complément à 1 du ET logique de n1 et n2.

NOR	(n1 n2 --- n3)	Complément à 1 du OU logique de n1 et n2.
NUMBER	(adr --- n,)	Convertit une chaîne de caractères à l'adresse adr en un nombre signé sur 32 bits, avec message d'erreur en cas d'impossibilité.
O.	(n ---)	Affiche un nombre en octal sans affecter la base courante. "o—dot"
OCTAL		Fixe la valeur de la base courante à 8.
OFFSET	(--- adr)	Variable qui contient la valeur à ajouter au numéro courant du bloc, pour déterminer son adresse physique réelle.
OTHERWISE		Voir la syntaxe IFTRUE...
PAGE		Efface l'écran du terminal, ou plus généralement effectue un 'form—feed'.
REMEMBER	syntaxe : REMEMBER <name>	Définit un mot qui lorsqu'il sera exécuté, détruira du dictionnaire <name> et toutes les définitions qui le suivent chronologiquement. ex : REMEMBER ICI .... ICI REMEMBER ICI
ROTATE	(n1 n2 --- n3)	Effectue une rotation des bits de n1, de n2 bits vers la gauche si n2>0, vers la droite si n2<0.
S0	(--- adr)	Retourne l'adresse du fond de pile (bottom of stack), "s—zero"
SET	(n adr ---) syntaxe : n adr SET <name>	Permet de définir des mots qui lorsqu'ils seront exécutés, rangeront la valeur n à l'adresse adr.
SHIFT	(n1 n2 --- n3)	Décalage logique de n2 bits dans le nombre n1. si n2>0, décalage gauche si n2<0, décalage droit. Les bits dégagés sont mis à zéro.
SP@	(--- adr)	Retourne l'adresse du sommet de la pile, juste avant l'exécution de SP@. "s—p—fetch"
TEXT	(c---)	Transfère des caractères de l'INPUT STREAM dans PAD. Les caractères restants de PAD sont mis à blanc.
THRU	(n1 n2 ---)	Charge consécutivement les blocs numérotés de n1 à n2.

U.R	(un1 n2 ---)	Affiche un1, non-signé sur 32 bits, justifié à droite dans un champs de n2 caractères. "u–dot–r"
USER	(n ---)	Permet de définir une USER VARIABLE <name>. n est l'offset par rapport au début de la USER AREA. L'exécution de <name> retourne sur la pile l'adresse absolue de le cellule où est rangée le valeur.
VLIST		Édite la liste des noms des mots du vocabulaire de CONTEXT, en commençant par la définition la plus récente.
WHERE		Affiche des mots d'état sur FORTH à savoir : — le dernier mot compilé — le dernier bloc utilisé
/LOOP	(n ---)	Mot de fin de structure DO LOOP. L'indice de boucle est décrémenté de n à chaque passage jusqu'à être inférieur à l'indice de fin. n doit être positif. "down–loop".

ANNEXE 2

ÉDITEUR

Nous avons vu dans le chapitre sur l'éditeur quelles étaient les commandes nécessaires et quelles étaient leurs fonctions. Un très bon exercice consiste à écrire un éditeur.

Dans cette annexe nous vous donnons un exemple d'éditeur que vous pourrez utiliser et perfectionner.

Les mots sont suffisamment simples pour que nous ne mettions pas de commentaires, ce qui vous permettra de vous familiariser avec la lecture d'un FORTH non commenté.

Les blocs ont une taille de 1024 octets (écran unitaire). Chaque écran comprend 16 lignes de 64 caractères.

Les blocs de texte sont gardés soit dans les DISK-BUFFERS, soit dans le SCREEN-BUFFER.

Le numéro d'écran correspond à l'adresse physique du bloc à l'écran sur la mémoire de masse. Sa valeur est gardée dans la USER VARIABLE SCR.

La position du curseur dans le SCREEN–BUFFER est gardée dans une autre USER VARIABLE R#.

Le texte destiné à être inséré ou supprimé du SCREEN BUFFER est temporairement conservé dans le TEXT BUFFER AREA, pointé par le mot PAD, qui retourne une adresse mémoire située 68 octets au-dessus du dictionnary pointer DP. PAD est utilisé comme "mémoire brouillon" pendant la mise au point de la ligne courante.

Tous les mots définis vont être regroupés dans le vocabulaire EDITOR, défini comme :

```
VOCABULARY EDITOR IMMEDIATE [Return] OK
```

Pour rendre EDITOR vocabulaire CURRENT, il suffit alors de taper :

```
EDITOR DEFINITIONS [Return] OK
```

Toutes les définitions suivantes seront ajoutées à EDITOR au lieu d'être traitées comme des définitions FORTH ordinaires.

Deux mots de base vont servir à réaliser les fonctions de mise au point de la ligne, TEXT et LINE.

TEXT déplace une ligne de texte de l'INPUT STREAM vers le TEXT BUFFER AREA du PAD.

LINE calcule l'adresse de la ligne dans le SCREEN BUFFER.

```
: TEXT          ( c ---        c est le caractère délimiteur)
    HERE        ( Adresse de début du WORD BUFFER où l'inter-)
                ( préteur de texte place les caractères)
    C/L 1+      ( nombre de caractères par ligne + 1)
    BLANKS      ( mise à blanc)
    WORD        ( déplace le texte, délimité par c, de l'INPUT)
                ( STREAM au WORD BUFFER)
    PAD         ( adresse du TEXT BUFFER)
    C/L 1+      ( nombre de caractères par ligne + 1)
    CMOVE       ( Déplace le texte, 64 octets+longueur, vers PAD)
; [Return] OK
```



```
: LINE          ( n --- adr   retourne l'adresse du début de la li-)
                ( gne n dans le SCREEN BUFFER. Charge le bloc s'il)
                ( n'est pas encore dans les DISC—BUFFER)
    DUP
    FFF0H  AND
    17  ?ERROR  ( s'assurer qu'il est bien compris entre 0 et 15)
    SCR         ( retourne le numéro de l'écran courant)
    (LINE)      ( Transfère l'écran du DISC BUFFER vers le)
                ( SCREEN BUFFER, et éventuellement charge un)
                ( bloc dans les DISC BUFFER.)
                ( Calcule l'adresse de la n-ième ligne du SCREEN)
                ( BUFFER, et la met sur la pile.)
    DROP
; [Return] OK
```

On peut alors mettre directement ces mots en pratique :

```
: -MOVE         ( adr n --- Copie une ligne de texte à partir de)
                ( l'adresse adr dans la n-ième ligne du SCREEN)
                ( BUFFER courant)
    LINE        ( calcule l'adresse de la ligne n)
    C/L  CMOVE  ( Déplace 64 caractères de adr vers la ligne n)
    UPDATE      ( Notifie au HANDLER de disque que le bloc a été)
                ( modifié)
; [Return] OK

: H             ( n --- Copie la n-ième ligne du SCREEN BUFFER)
                ( dans PAD. Le texte est alors prêt à être affiché)
    LINE        ( calcule l'adresse de la ligne)
    PAD  1+     ( adresse de début de PAD)
    C/L  DUP    ( nombre de caractères par ligne)
    PAD  C!     ( met la longueur de PAD à 64)
    CMOVE       ( déplace la n-ième ligne)
; [Return] OK
```

```
. S              ( n ––– Met à blanc la n-ième ligne, décale vers)
                 ( le bas l'ancienne ligne et toutes les suivantes une)
                 ( par une. La dernière ligne est perdue)
   DUP  1–       ( dernière ligne à déplacer)
   0EH           ( 14, numéro de la dernière ligne à décaler)
   DO
   I  LINE       ( calcule l'adresse de la n-ième ligne)
   I  1+         ( numéro de ligne suivante)
   –MOVE         ( décalage d'une ligne vers le bas)
   –1  +LOOP     ( décrémente le compteur de boucle de 1)
   E             ( efface la n-ième ligne)
; [Return] OK

: D              ( n ––– Supprime la n-ième ligne du SCREEN)
                 ( BUFFER et remonte toutes les lignes consécu-)
                 ( tives de 1 cran. La ligne supprimée reste dans)
                 ( PAD.)
   DUP  H        ( copie de la n ième ligne dans PAD)
   0FH           ( numéro de la dernière ligne)
   DUP  ROT
   DO
   I  1+  LINE   ( adresse de la ligne à déplacer)
   I  –MOVE      ( déplacement de un vers le haut)
   LOOP
   E             ( effacer la dernière ligne)
; [Return] OK
```

Il est possible d'écrire également les mots élémentaire de manipulation de la ligne courante :

```
: E              ( n ––– Efface la n-ième ligne, en la remplissant)
                 ( de 64 caractères blancs.)
   LINE          ( adresse de la ligne)
   C/L  BLANKS   ( mise à blanc)
   UPDATE        ( marquage du buffer)
; [Return] OK
```



```
: R              ( n ––– Remplace la n-ième ligne par le texte)
                 ( contenu dans PAD.)
   PAD  1+       ( adresse de début de PAD)
   SWAP – MOVE   ( copie le texte depuis PAD dans la n-ième ligne)
; Return OK

: P              ( n ––– Place le texte qui suit dans la n-ième ligne.)
                 ( l'ancien texte est perdu.)
   1  TEXT       ( accepte le texte qui suit, avec deux conditions)
                 ( d'arrêt : C/L caractères ou rencontre de CR, dans)
                 ( la n-ième ligne)
   R
; [Return] OK

: I              ( n ––– Insère un texte dans le SCREEN BUFFER,)
                 ( initialement contenu dans PAD. Toutes les lignes)
                 ( qui suivent la ligne n sont décalées vers le bas)
                 ( et la dernière est perdue)
   DUP  S        ( met la n-ième ligne à blanc)
   R             ( Copie le contenu de PAD dans la n ième ligne)
; [Return] OK
```

Voici les mots qui travaillent sur des écrans entiers :

```
: CLEAR          ( n ––– Efface le n-ième écran en mettant toutes)
                 ( ses lignes à blanc)
   SCR  !        ( met à jour SCR)
   10H  0  DO
   FORTH  I      ( Passage en FORTH pour avoir la bonne définition)
                 ( de I, qui donne l'indice courant de la boucle.)
   EDITOR  E     ( effaçage de la ligne)
   LOOP
; [Return] OK
```

```
COPY                ( n1 n2 ——— copie de l'écran n1 sur l'écran n2)
    B/SCR *         ( premier bloc correspondant à l'écran n2)
    OFFSET @ +      ( adresse physique sur le disque)
    SWAP            ( adr n1 ——— )
    B/CSR *         ( premier bloc correspondant à l'écran n1)
    B/SCR OVER +    ( adr bloc2 bloc2+1 ——— )
    SWAP            ( adr(n2) fin(n1)+1 debut(n1)+1 ———)
    DO              ( adr(n2) ———)
    DUP             ( adr(n2) adr(n2))
    FORTH I         ( adr(n2) adr(n2) adr—courante(n1))
    BLOCK           ( charge le bloc courant de l'écran n1 dans le)
                    ( DISC BUFFER et retourne son adresse)
    2— !            ( lui donne comme numéro, le numéro du bloc)
                    ( courant dans l'écran n2)
    1+              ( passe au bloc suivant dans l'écran n2)
    UPDATE          ( signale la modification du bloc)
    LOOP
    DROP
    FLUSH           ( mise à jour sur disque de tous les blocs)
                    ( modifiés)
; [Return] OK
```

Tous les mots présentés ci-dessus constituent un petit éditeur de page ligne par ligne, au sens que l'objet élémentaire susceptible d'être modifié est la ligne de texte.

Les mots que nous allons voir maintenant permettent de travailler la ligne elle-même au niveau de la chaîne de caractères.

La variable R# contient un curseur pointant sur le prochain caractère qu'il est possible d'atteindre par l'éditeur de chaîne.

Le tout premier mot à définir est celui qui permet de chercher une chaîne dans un texte, en déplaçant le curseur au fur et à mesure :


```
: MATCH                     ( ad1 n1 ad2 n2 ——— f n3 )
    >R >R 2DUP              ( duplication de ad1 et n1 )
    R> R> 2SWAP             ( ad1 n1 ad2 n2 ad1 n1 ——— )
    OVER + SWAP             ( ad1 n1 ad2 n2 ad1+n1 ad1 ——— )
    DO                      ( ad1 n1 ad2 n2 ——— )
    2DUP                    ( ad1 n1 ad2 n2 ad2 n2 ——— )
    FORTH I                 ( ad1 n1 ad2 n2 ad2 n2 I ——— )
    —TEXT                   ( le texte (2) et le texte (1) correspondent-ils)
                            ( sur les n2 premiers caractères ?)
    IF                      ( ad1 n1 ad2 n2 ——— )
    >R 2DROP R>             ( ad1 n2 ——— )
    — I SWAP —              ( n3 ——— )
    0 SWAP                  ( 0 n3 ——— )
    0 0 LEAVE               ( quitter la boucle avec 2 zéro muets)
    THEN
    LOOP
    2DROP                   ( nettoyer la pile)
    SWAP 0= SWAP            ( mettre le flag dans le bon sens)
; [Return] OK

: —TEXT                     ( ad1 n ad2 ——— f  Si les chaînes com-)
                            ( mençant à ad1 et ad2 coïncident sur les n)
                            ( premiers caractères, retourne un flag)
                            ( TRUE, sinon retourne un flag FALSE)
    SWAP                    ( ad1 ad2 n ——— )
    —DUP IF                 ( si n vaut zéro, évite le traitement)
    OVER + SWAP             ( ad1 ad2+n ad2 ——— )
    DO                      ( ad1 ——— )
    DUP C                   ( ad1 c ——— )
    FORTH I C —             ( ad1 c—c' ——— )
    IF 0= LEAVE             ( ils sont différents, on quitte la bouche )
    ELSE 1+                 ( sinon on continue à comparer )
    THEN
    LOOP
```

On peut alors écrire les premiers mots de manipulation de la ligne elle-même :

```
: TOP              ( positionne le curseur sur le premier carac-)
                   ( tère de la première ligne de l'écran courant)
   0 R#  !
; [Return] OK

: #LOCATE          ( ——— n1 n2 A  partir de la position courante)
                   ( du curseur, calcule le numéro de la ligne et)
                   ( l'offset du curseur par rapport au début de)
                   ( la ligne)
   R#
   C/L  /MOD
; [Return] OK

: #LEAD            ( ——— adr n  A partir de la position courante)
                   ( du curseur, calcule l'adresse adr de la ligne)
                   ( dans le SCREEN BUFFER, et l'offset par)
                   ( rapport au début de la ligne.)
   #LOCATE
   LINE
   SWAP
; [Return] OK

: #LAG             ( ——— adr n  A partir de la position courante)
                   ( du curseur, calcule l'adresse de la ligne)
                   ( dans le SCREEN BUFFER, et l'offset par)
                   ( rapport à la fin de la ligne.)
   #LEAD           ( paramètres par rapport au début de la ligne)
   DUP  >R         ( sauvegarde l'offset)
   +  C/L  R>  —
; [Return] OK
```



```
: M                ( n ——— Avance le curseur de n caractères.)
                   ( Affiche la ligne y compris le curseur pour)
                   ( modifications)
   R#  +!          ( déplacement du curseur)
   CR  SPACE       ( nouvelle ligne)
   #LEAD  TYPE     ( affichage du texte qui précède le curseur)
   5FH  EMIT       ( affichage du curseur '↑')
   #LAG  TYPE      ( affichage du texte qui suit le curseur)
   #LOCATE  .      ( affichage du numéro de ligne)
   DROP
; [Return] OK

: T                ( n ——— Affiche le contenu de la n-ième ligne)
                   ( et en fait une copie dans PAD)
   DUP  C/L  *     ( calcule l'offset de la n ième ligne dans)
                   ( l'écran)
   R#  !           ( pointe le curseur sur le début de la n ième)
                   ( ligne)
   H               ( copie la n ième ligne dans PAD)
   0  M            ( édite la n ième ligne sur le périphérique de)
                   ( sortie)
; [Return] OK

: L                ( Affiche à l'écran tout le contenu de l'écran)
                   ( en cours de modification)
   SCR @  LIST     ( liste l'écran courant)
   0  M            ( affiche la ligne contenant le curseur)
; [Return] OK

: 1LINE            ( ——— f  Recherche dans la LIGNE cou-)
                   ( rante à partir de la position courante)
                   ( du curseur, une chaîne rangée dans PAD)
   #LAG  PAD  COUNT( prépare les paramètres de MATCH)
   MATCH

; [Return] OK
```

```
: FIND                 ( Recherche dans l'écran entier une)
                       ( chaîne rangée dans PAD. Affiche un )
                       ( message d'erreur en cas d'échec,)
                       ( sinon repositionne le curseur)
   BEGIN
   3FFH  R#  @  <      ( teste la curseur par rapport à 1023)
   IF                  ( si en dehors de l'écran)
   TOP                 ( en haut à gauche)
   PAD  HERE  C/L  1+  ( déplace la chaîne vers HERE pour)
    CMOVE              ( l'afficher dans le message d'erreur)
   0  ERROR
   ENDIF
   1LINE               ( cherche dans la ligne)
   UNTIL
; Return OK

: DELETE               ( n ——— Supprime n caractères à partir de la)
                       ( position du curseur et recompacte le texte)
                       ( dans la ligne)
   >R                  ( sauvegarde le nombre de caractères à)
                       ( supprimer)
   #LAG  +             ( fin de ligne)
   FORTH  R  –         ( R copie le sommet de la pile des return)
   #LAG
   R  MINUS  R#  +!    ( reculer le curseur de n)
   #LEAD  +            ( nouvelle position du curseur)
   SWAP  CMOVE         ( déplace le reste de la ligne pour la)
                       ( suppression)
   R>  BLANKS          ( met des blancs aux emplacements libérés)
   UPDATE              ( mise à jour du BUFFER)
; Return OK
```



```
: N                    ( Trouve la prochaine occurence d'une chaîne déjà)
                       ( dans PAD)
   FIND                ( recherche)
   0  M                ( affichage de la ligne en cas de succès)
; Return OK

: F                    ( Trouve la première occurrence de la chaîne qui suit)
                       ( dans l'INPUT STREAM)
   1  TEXT
   N
; Return OK

: B                    ( Ramène le curseur au début de la chaîne qui coïn-)
                       ( cide avec une chaîne donnée)
   PAD  C@             ( Ramène la longueur de la chaîne contenue dans)
                       ( PAD)
   MINUS  M            ( Recule le curseur et réaffiche la ligne)
; Return OK

: X                    ( Supprime le texte qui suit dans l'INPUT STREAM)
   1  TEXT             ( transporter dans PAD)
   FIND                ( chercher dans l'écran)
   PAD  C@             ( longueur de la chaîne à supprimer)
   DELETE              ( suppression)
   0  M                ( affichage)
; Return OK
```

```
:                        ( Supprime tous les caractères depuis la po-)
                         ( sition courante du curseur jusqu'y compris)
                         ( la première occurence du texte qui suit)
                         ( dans l'INPUT STREAM)
    #LEAD  +             ( adresse courante du curseur)
    1  TEXT              ( transport dans PAD)
    1LINE                ( recherche dans la ligne)
    0=  0  ?ERROR        ( erreur si elle n'existe pas)
    #LEAD  +  SWAP  -    ( preparer les paramètres pour DELETE)
    DELETE
    0  M
    ;  [Return]  OK
```

```
:  C                               ( Insertion dans le texte à la position)
                                   ( courante du curseur, de carac-)
                                   ( tères qui suivent dans l'INPUT)
                                   ( STREAM. Les caractères qui sor-)
                                   ( tent de la ligne sont perdus.)
1  TEXT  PAD  COUNT                ( transport dans PAD)
#LAG  ROT  OVER  MIN  >R           ( sauvegarder MIN(nb—car, pos—)
                                   ( curseur)
FORTH  R                           ( copie de la pile des return)
R#  +!                             ( repositionner le curseur)
R  —  >R                           ( nombre de caractères à garder)
DUP  HERE  R  CMOVE                ( déplacement temporaire à HERE)
HERE  #LEAD  +  R>  CMOVE          ( Ramener le texte)
     R>  CMOVE                     ( replacer la chaîne)
     UPDATE
     0  M
;  [Return]  OK
```


ANNEXE 3
SOLUTION DES EXERCICES

CHAPITRE 2:

1. a) (———) La pile est vide.
b) (23 33 54 ———)
c) (32 15 ———)
d) (13 15 18 ———)

2. a) ab+cd+*
b) ab+c/de+f*+
c) abc+/de+ac/*/
d) abc/*da+bcd+/*+

CHAPITRE 3:

1. a) (2 3 1 ———)
b) (2 5 ———)
c) (6 ———)
d) (–20 ———)
e) Erreur : pile vide
f) (1 2 3 –1 3 –1 ———)

2. a)
```
: EXO2A  OVER  +  SWAP  2SWAP  ROT  ROT  +  *  SWAP  /  ;
```
b)
```
: EXO2B  >R  >R  ROT  OVER  +  ROT  R>+  OVER  *
         R>  SWAP  OVER  /  ROT  2SWAP  +  *  SWAP  /;
```
c)
```
: EXO2C  2DUP  +  >R  ROT  DUP  >R  +  R>  OVER  *  R>
         2SWAP  *  +SWAP  /  ;
```

3. a)

```
: MINIMUM                  ( ad n --- min )
    7FFF  SWAP             ( ad 7FFF n --- )
    0 DO                   ( ad min --- )
        OVER               ( ad min Val(ad) --- )
        MIN                ( ad min' --- )
        SWAP 2+ SWAP       ( ad+2 min' --- )
    LOOP
    SWAP  DROP             ( min --- )
;
```

b)

```
: MAXIMUM                  ( ad n --- max )
    FFFF  SWAP             ( ad FFFF n --- )
    0 DO                   ( ad max --- )
        OVER               ( ad max val(ad) --- )
        MAX                ( ad max' --- )
        SWAP 2+ SWAP       ( ad+2 max' --- )
    LOOP
    SWAP  DROP             ( max --- )
;
```

4. a)

```
: SOUSTRACTION  MINUS  +  ;
```

b)

```
: */MOD  >R  U*  R>  U/MOD  ;
```

5. a)

```
: 0FILL                    ( ad n --- )
    0 DO                   ( ad --- )
        0  OVER            ( ad 0 ad --- )
        !                  ( ad --- )
        2+                 ( ad+3 --- )
    LOOP
    DROP                   ( --- )
;
```

b)

```
: +1FILL                   ( ad n --- )
    0 DO                   ( ad --- )
        1  OVER            ( ad 1 ad --- )
        +!                 ( ad --- )
        2+                 ( ad+2 --- )
    LOOP
    DROP                   ( --- )
```

6. a) Nous allons utiliser pour résoudre cet exercice l'algorithme d'Euclide qui s'écrit de la manière suivante :

Soit à calculer le PGCD des nombres a et b. On effectue :

$$a = b.q + r \quad \text{(division entière)}$$

On sait que r est inférieur à b. On peut donc écrire :

$$b = r.q1 + r1$$

et ainsi de suite, jusqu'à obtenir :

$$r_{n-1} = r_n\, q_{n+1} + r_{n+1}$$

avec $r_{n+1} = 0$

On sait que le PGCD est alors r_n.

```
: PGCD
   SMUDGE
   DUP  IF
          SWAP  OVER
          /MOD  DROP
          PGCD
        ELSE
          DROP
        THEN
   SMUDGE;
```

b) Pour le calcul du PPCM, on utilisera le résultat suivant :

$$A * B = PGCD(A,B) * PPCM(A,B)$$

```
: DIV1
     DUP  >R  /MOD
     .  R>  OVER
     IF
       46  EMIT
       4  0  DO  SWAP  10  *
                 OVER  /MOD
                 48  +  EMIT
                 DUP  IF
                      ELSE  LEAVE
                      THEN
                 SWAP
                 LOOP
     THEN
     DROP  DROP
;
```

8.

```
: DIV2                              ( a b --- )
    0                               ( a b n --- )
    BEGIN                           (nbr décimales)
        OVER  >R  >R                ( a b --- )
                                    ( b n --- )
        /MOD                        ( r q --- )
        DUP  IF
            I  1  =
            IF
                46  EMIT            (Point décimal)
            THEN
            48  +  EMIT             ( r --- )
                                    ( b n --- )
            R>  R>                  ( r n b --- )
            SWAP  1+                ( r b n+1---)
            0                       ( r b n+1
                                    false --- )
        ELSE
            DROP  DROP  R>  R>  DROP    ( n --- )
            IF  ELSE  48  EMIT  THEN    ( Affichage 0 )
            1                           ( True --- )
        THEN
    UNTIL
;
```

9.

```
: DIV3                              ( a b --- )
    0                               ( a b n --- )
                                    ( nbre décimales )
    BEGIN
        OVER  >R  >R                ( a b --- )
                                    ( b n --- )
        /MOD                        ( r q --- )
                                    ( b n --- )
        IF
            0                       ( r q false --- )
                                    ( b n --- )
        ELSE
            DROP DROP R> R> DROP    ( n --- )
            IF  ELSE  48  EMIT  THEN    ( Affichage 0 )
            1                       ( true --- )
        THEN
    WHILE
        I 1 =                       ( r q flag --- )
                                    ( b n --- )
        IF  46  EMIT  THEN          ( Point décimal )
        48  +  EMIT
        R>  R>                      ( r n b --- )
```


10.

```
: CONV                              ( n --- )
    16  0  DO
        I  15  -  2                 ( n 15-1 2 --- )
        SWAP  PUISS                 ( n 2puiss(15-1)
                                    --- )
        OVER  AND                   ( n flag --- )
        0=  0=                      ( normalisation flag )
        48  +  EMIT                 ( n --- )
    LOOP
    DROP
;
```

Le mot PUISS peut être défini comme suit :

```
: PUISS                             ( a b ---
                                    apuiss(b) )
    DUP  0=
    IF
        DROP  DROP  1               ( 1 --- )
        ELSE
        1  SWAP                     ( a 1 b --- )
        1  DO                       ( a 1 --- )
          OVER  *                   ( a apuiss(1) --- )
        LOOP                        ( a apuiss(b) --- )
        SWAP  DROP                  ( apuiss(b) --- )
    THEN
;
```

11. Nous allons utiliser pour ce faire la méthode du crible d'Eratostène. Elle consiste à construire un vecteur contenant tous les nombres de 1 à 100. On parcourt ce tableau de 1 à 100. Lorsque l'on trouve un nombre valide, on le sort comme étant premier, et l'on invalide du vecteur tous ses multiples. La méthode d'invalidation consistera ici à remplacer la valeur par 0. Dans notre vecteur, nous stockerons les valeurs sur 1 octet.

```
0 VARIABLE ERATOS 98 ALLOT

        ERATOS                  ( Pfa(ERATOS) --- )
        100  0  DO
            I  OVER  C!         ( Ad -)
            1+
        LOOP
        DROP
;

SET ERATOS
```

Construisons maintenant le mot N–MUL qui invalidera de ERATOS tous les multiples de n.

```
: N–MUL                        ( n ––– )
      101   OVER               ( n 101 n ––– )
      DO
          I  1  –  ERATOS      ( n i–1 ad ––– )
          +                    ( n ad(i) ––– )
          0  SWAP  C!          ( n ––– )
          DUP
      +LOOP
      DROP
;

: PREMIER                      ( ––– )
      ERATOS  1  .             ( ad ––– )
      101  3  DO
         I  1  –               ( ad i–1 ––– )
         OVER  +  C            ( ad n ––– )
         DUP  IF
             DUP  .  N–MUL     ( ad –––)
         ELSE
             DROP              ( ad ––– )
         THEN
      LOOP
      DROP                     ( ––– )
;
```

12. Définissons d'abord deux vecteurs :

```
0 VARIABLE
ALPHABET
    25  ALLOT
```

ainsi qu'un mot pour les remplir de lettres :

```
: MET
    [COMPILE]  '  ( Pfa ––– )
    32  WORD
    HERE  COUNT ( Pfa Hereti Long ––– )
    >R  SWAP  R> ( Pfa Here+1 Long ––– )
    CMOVE
;
```


Définissons le mot POSITION qui recherche une chaîne de caractères dans une autre. Il faut lui fournir l'adresse et la longueur de la chaîne recherchée et l'adresse et la longueur de la chaîne cible.

```
: TESTE          ( ad n ad-c n-c ––– f )
  SMUDGE
  ROT            ( ad ad-c n-c n ––– )
  2DUP  >=  IF
          ROT  ROT
          2OVER  2OVER
          DROP  –TEXT ( ad n ad-c n-c f ––– )
              IF
                ROT  2DUP
                >  IF       ( ad ad-c n-c n ––– )
                     ROT  1+
                     ROT  –1    ( ad n ad-c+1 n-c–1 ––– )
                     TESTE
                ELSE
                     2DROP
                     2DROP  0
                THEN
              ELSE
                DROP  ROT
                ROT  2DROP       ( ad' ––– )
              THEN
        ELSE
              2DROP  2DROP  0      ( 0 ––– )
        THEN
  SMUDGE
;

: MODIF
     1  ALPHABET  27  TESTE
     DUP  IF  ALPHABET –
              DUP  6  <  IF
                            1  VOY  +!
                            DROP
                         ELSE
                            26  <  IF
                                      1  CON  +!
                                   ELSE
                                      1  BLA  +!
                                   THEN

          ELSE
              DROP
          THEN
```

```
: COMPTE                          ( ––– )
      0  VOY  !  0  CON  !  0  BLA  !
      TIB @ IN @ +
      BEGIN
           DUP  C@
      WHILE
           DUP  MODIF  1+
      REPEAT
      DROP  VOY  ?  CON  ?  BLA  ?
      QUIT
;
```

CHAPITRE 6 :

1. a)

```
: ,                      ( n –––)
     HERE  !
     2  ALLOT
;
```

b)

```
: C,                     ( n ––– )
     HERE  C!
     1  ALLOT
;
```

2. Le but du mot COMPILE est de différer la compilation au moment de l'exécution. Le cfa du mot qui suit COMPILE est alors placé dans la première cellule libre du dictionnaire. On se rappelle de plus que, au moment de l'exécution de COMPILE, le pointeur d'interprétation se trouve sur la pile de Return.

```
: COMPILE
     R>                  ( Pointe sur le mot qui suit )
     DUP  2+  >R         ( Incrémente IP )( Ad ––– )
     @ ,                 ( Compile le cfa du mot )
;
```

3

```
: ..                             ( Ad ––– )
     COMPILE  OBRANCH            ( Pour le déroutement )
     HERE                        ( Pour calcul de l'offset )
     0 ,                         ( Réservation cellule )
;
IMMEDIATE

: THEN                           ( Ad ––– )
     HERE                        ( Adr. absolue déroutement )
     OVER  –                     ( Adr relative déroutement )
     SWAP  !                     ( Stocke offset OBRANCH ou
                                               BRANCH )
;
IMMEDIATE

: ELSE                           ( Adr ––– Adr' )
     COMPILE  BRANCH             ( Pour déroutement )
     HERE  0 ,                   ( Adr Here ––– )
     SWAP                        ( Here Adr ––– )
     [  COMPILE  ]  THEN         ( THEN est immédiat ! )
;
IMMEDIATE
```

— Stevens W.R. *A FORTH Primer*
Kitt Peak National Observatory, 1979

— Taylor A. *FORTH Becoming Hothouse for Developping Languages*
Computerworld, July 1979

— Ting C.H. *System-Guide to Fig-FORTH*
Offete Enterprises Inc.

— Wells D.C. *Interactive Image Analysis for Astronomers*
Computer, August 1977

INDEX

Imprimé en France. — JOUVE, 18, rue Saint-Denis, 75001 PARIS

Dépôt légal : Août 1983
N° d'Éditeur : 3991